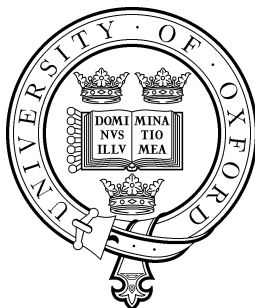


THE SAFE LAMBDA CALCULUS

William Blum

Linacre College

*Submitted in partial fulfilment of the requirements for
the degree of Doctor of Philosophy*



Oxford University Computing Laboratory

Draft of August 12, 2008— Michaelemas 2008

Abstract

We consider a syntactic restriction for higher-order grammars called *safety* that constrains occurrences of variables in the production rules according to their type-theoretic order. We transpose and generalize this restriction to the setting of the simply typed lambda calculus, giving us what we call the *safe lambda calculus*. We study the expressivity of the calculus and show a result in the same vein as Schwichtenberg's 1976 characterization of the simply typed lambda calculus: we show that the numeric functions representable in the safe lambda calculus are exactly the multivariate polynomials; thus conditional is not definable. We also give a characterization of representable word functions. We then study the complexity of deciding beta-eta equality of two safe simply typed terms and show that this problem is PSPACE-hard. The safety restriction is then extended to other applied lambda calculi featuring recursion and references such as PCF and Idealized Algol.

In order to study the game semantics of safe languages, we introduce a new concrete presentation of game semantics based on the theory of *traversals*: We show that the *revealed game denotation* of a term can be computed by traversing some souped-up version of the abstract syntax tree of the term using adequately defined traversal rules. This result was presented at the Galop workshop at ETAPS 2008. This allows us to give a game-semantic analysis of safety via syntactic reasoning: we show that safe lambda-terms are denoted by what we call *P-incrementally justified strategies*. This result was presented at TLCA 2007.

We then study models of the safe lambda calculus and show that these are captured by *Incremental Closed Categories*. We then build a categorical game model of the safe lambda calculus which gives rise to a fully abstract game model of safe IA. The model obtained for safe IA is effectively presentable: two terms are equivalent just if they have the same set of complete *O-incrementally justified* plays, where O-incremental justification is defined as the dual of P-incremental justification.

Finally in the last chapter we study safety from the point of view of Algorithmic Game Semantics. We observe that up to the 3rd order, the addition of unsafe context is conservative for observational equivalence (for both IA and safe IA). This implies that all the upper complexity bounds known for the lower-order fragments of IA are also valid for the safe fragment; we show that it is also the case for the lower-bounds. At order 4, observational equivalence was shown to be undecidable for IA. We conjecture that for the order-4 *safe* fragment of IA, the problem is reducible to the DPDA-equivalence problem (which is decidable).

Contents

1	Introduction	1
2	Background	7
2.1	Lambda Calculus	7
2.1.1	Terms	7
2.1.2	Substitution	8
2.1.3	Conversion	9
2.1.4	Properties	10
2.1.5	Simple types	10
2.1.6	Simply typed lambda calculus <i>à la</i> Curry	10
2.1.7	Simply typed lambda calculus <i>à la</i> Church	12
2.1.8	Extensions	13
2.1.9	PCF	13
2.1.10	Idealized Algol	15
2.2	Higher-Order Grammars and the Safety Restriction	18
2.2.1	Higher-order grammars	18
2.2.2	The safety restriction	20
2.2.3	Automata Characterization	21
2.2.4	Expressivity	22
2.2.5	Is safety a genuine restriction?	22
2.2.6	Higher-order grammars and the simply typed lambda calculus	22
2.3	Game Semantics	24
2.3.1	History	24
2.3.2	Definitions	25
2.3.3	On the necessity of justification pointers	31
2.3.4	Categorical interpretation	34
2.3.5	The fully abstract game model of PCF	36
2.3.6	The fully abstract game model of Idealized Algol	41
2.3.7	Algorithmic game semantics	43
3	The Safe Lambda Calculus	45
3.1	Definition and properties	46
3.1.1	Safety adapted to the lambda calculus	46
3.1.2	Safe beta reduction	51
3.1.3	Eta-long normal form	54
3.1.4	Almost safety	56
3.1.5	Safety with respect to other type-ranking functions	58
3.1.6	Homogeneous safe lambda calculus	58
3.2	Complexity	61
3.2.1	Statman's result	62
3.2.2	Mairson's encoding	62
3.2.3	PSPACE-hardness	65

3.2.4	Other complexity results	66
3.3	Expressivity	67
3.3.1	Numeric functions representable in the safe lambda calculus	67
3.3.2	Word functions definable in the safe lambda calculus.	69
3.4	Typing problems	73
3.4.1	Relating derivations from $\Lambda_{\rightarrow}^{\text{Cu}}$ and safe $\Lambda_{\rightarrow}^{\text{Cu}}$	73
3.4.2	Type checking and typability	74
3.4.3	The type inhabitation problem	75
3.5	Extensions	76
3.5.1	PCF	76
3.5.2	Idealized Algol	80
3.5.3	Generalization to other applied lambda calculi	86
3.6	Related work	86
4	A Concrete Presentation of Game Semantics	89
4.1	Computation tree	90
4.1.1	Definition	90
4.1.2	Pointers and justified sequence of nodes	93
4.1.3	Traversal of the computation tree	96
4.2	Game semantics correspondence	116
4.2.1	Interaction game semantics	116
4.2.2	Relating computation trees and games	124
4.2.3	Mapping traversals to interaction plays	128
4.2.4	The correspondence theorem for the pure simply typed lambda calculus	130
4.3	Extension to PCF and IA	140
4.3.1	PCF fragment	141
4.3.2	Idealized algol	146
4.4	Conclusion and related works	151
5	Syntactic Analysis of the Game Denotation of Safe Terms	153
5.1	P-incrementally justified strategies	154
5.2	Dead code elimination	154
5.3	Incremental binding	155
5.4	Safe lambda calculus	159
5.5	Safe PCF	161
5.6	Safe Idealized Algol	162
5.7	Towards a game model of safe PCF	163
5.7.1	Definability	163
5.7.2	Compositionality	163
5.7.3	Full abstraction	165
6	Models of Safe Applied Lambda Calculi	167
6.1	Categorical model	167
6.1.1	Safe lambda calculus with product	167
6.1.2	Incremental closed category	170
6.1.3	Categorical semantics	172
6.1.4	Quotiented category	173
6.1.5	The internal language of incremental closed categories	173
6.2	The game model	174
6.2.1	Order of a move	175
6.2.2	Well-bracketing	175
6.2.3	P-incremental justification	177
6.2.4	Closed P-incremental justification	177
6.2.5	Interaction sequences	178

6.2.6	Preliminary results	180
6.2.7	Categories of closed P-i.j. strategies	188
6.3	Interpretation in the standard game model	189
6.3.1	Safe lambda calculus with product	189
6.3.2	Safe PCF	189
6.3.3	Safe Idealized Algol	190
6.4	O-incremental justification	193
6.5	Full abstraction	194
6.6	Algorithmic game semantics	198
7	Conclusion	203
7.1	Summary of contribution	203
7.2	Further works	204
	Bibliography	209
	Index to Notations	215
	Index	217

List of Figures

2.1	Strategy denotation of the case construct.	40
4.1	Tree-representation of the revealed strategy in the application case.	122
4.2	Flow-diagram for interaction plays of $\langle\langle \Gamma \vdash x_i N_1 \dots N_p \rangle\rangle$	124
4.3	Example of a sequence $u \upharpoonright A, B, C$ for $u \in \langle\langle M \rangle\rangle_s$ and $l = 1$	137
4.4	Transformations involved in the Correspondence Theorem.	140
4.5	Computation tree of $\lambda xy. \text{cond } 1 \ x \ y$	145
6.1	Structure of an interaction sequence.	179
6.2	State diagram for plays of σ^\dagger	187

List of Tables

2.1	Formation rules for PCF terms.	14
2.2	Big-step operational semantics of PCF.	14
2.3	Formation rules for IA.	16
2.4	Big-step operational semantics of IA.	17
2.5	Algorithm LmdToHOGr.	23
2.6	The complete complexity classification for observational equivalence in IA.	44
3.1	The safe lambda calculus <i>à la</i> Curry.	46
3.2	Typing rules for long-safe terms-in-contexts.	54
3.3	Alternative Curry-style definition of the safe lambda calculus.	57
3.4	Alternative definition of the Curry-style lambda calculus	73
3.5	Formation rules for Safe PCF.	77
3.6	Formation rules for strongly safe IA	81
3.7	Formation rules for safe IA.	83
4.1	The tree $\tau^-(M)$	91
4.2	Type of the enabler node.	93
4.3	Traversal rules for the simply typed lambda calculus.	96
4.4	Computation hyper-trees of IA's constructs.	148
4.5	Traversal rules for IA constants.	149
6.1	The safe lambda calculus with product.	169
6.2	Complexity of observational equivalence in safe IA.	201
6.3	Murawski representability.	202

Chapter 1

Introduction

Background

The *safety condition* was introduced by Knapik, Niwiński and Urzyczyn at FoSSaCS 2002 [KNU02] in a seminal study of the algorithmics of infinite trees generated by higher-order grammars. The idea, however, goes back some twenty years to Damm [Dam82] who introduced an essentially equivalent¹ syntactic restriction (for generators of word languages) in the form of *derived types*. A higher-order grammar (that is assumed to be *homogeneously typed*) is said to be *safe* if it obeys certain syntactic conditions that constrain the occurrences of variables in the production (or rewrite) rules according to their type-theoretic order. Though the formal definition of safety is somewhat intricate, the condition itself is manifestly important. As we survey in the following, higher-order *safe* grammars capture fundamental structures in computation, offer clear algorithmic advantages, and lend themselves to a number of compelling characterizations:

- *Word languages.* Damm and Goerdt [DG86] have shown that the word languages generated by order- n *safe* grammars form an infinite hierarchy as n varies over the natural numbers. The hierarchy gives an attractive classification of the semi-decidable languages: Levels 0, 1 and 2 of the hierarchy are respectively the regular, context-free, and indexed languages (in the sense of Aho [Aho68]), although little is known about higher orders.

Remarkably, for generating word languages, order- n *safe* grammars are equivalent to order- n pushdown automata [DG86], which are in turn equivalent to order- n indexed grammars [Mas74, Mas76].

- *Trees.* Knapik et al. have shown that the Monadic Second Order (MSO) theories of trees generated by *safe* (deterministic) grammars of every finite order are decidable².

They have also generalized the equi-expressivity result due to Damm and Goerdt [DG86] to an equivalence result with respect to generating trees: A ranked tree is generated by an order- n *safe* grammar if and only if it is generated by an order- n pushdown automaton.

- *Graphs.* Caucal [Cau02] has shown that the MSO theories of graphs generated³ by *safe* grammars of every finite order are decidable. In a recent paper [HMOS08], however, Hague et al. have shown that the MSO theories of graphs generated by order- n *unsafe* grammars are undecidable, but deciding their modal mu-calculus theories is n -EXPTIME complete.

¹See de Miranda's thesis [dM06] for a proof.

²It has recently been shown [Ong06a] that trees generated by *unsafe* deterministic grammars (of every finite order) also have decidable MSO theories. More precisely, the MSO theory of trees generated by order- n recursion schemes is n -EXPTIME complete.

³These are precisely the configuration graphs of higher-order pushdown systems.

Overview

The aim of this thesis is to understand the safety condition in the setting of the typed lambda calculus. Our first task is to transpose it to the lambda calculus and pin it down as an appropriate sub-system of the simply-typed theory. A first version of the *safe lambda calculus* has appeared in an unpublished technical report [AdMO04]. Here we propose a more general and cleaner version where terms are no longer required to be homogeneously typed. The formation rules of the calculus are designed to maintain a simple invariant: Variables that occur free in a safe lambda-term have orders no smaller than that of the term itself. We can now explain the sense in which the safe lambda calculus is safe by establishing its salient property: No variable capture can ever occur when substituting a safe term into another. In other words, in the safe lambda calculus, it is *safe* to use capture-*permitting* substitution when performing β -reduction.

There is no need for new names when computing β -reductions of safe lambda-terms, because one can safely “reuse” variable names in the input term. Safe lambda calculus is thus cheaper to compute in this naïve sense. Intuitively one would expect the safety constraint to lower the expressivity of the simply typed lambda calculus. Our next contribution is to give a precise measure of the expressivity deficit of the safe lambda calculus. An old result of Schwichtenberg [Sch76] says that the numeric functions representable in the simply typed lambda calculus are exactly the multivariate polynomials *extended with the conditional function*. In the same vein, we show that the numeric functions representable in the safe lambda calculus are exactly the multivariate polynomials. We further obtain a similar characterization concerning representable word-functions.

In order to get a better understanding of our calculus, it is interesting to recast common problems studied in the literature on the simply typed lambda calculus in the setting of the safe lambda calculus. We show for instance that the type-checking and typability problems remain decidable. We also consider the type-inhabitation problem: “Is there a term inhabiting a given type?”. This problem is already relatively complex in the simply-typed lambda calculus—Statman showed that it is PSPACE-complete. Because of the somewhat intricate way in which safety constrains the occurrences of the variables, the inhabitation problem becomes even more complex in the safe lambda calculus. We do not know whether the problem is decidable.

Another famous result by Statman is that deciding beta-equality of two simply typed terms is non-elementary. There are several proofs of this results in the literature. All of them proceed by reduction of a non-elementary problem—such as quantifier elimination in finite type theory—into the simply typed lambda calculus. Interestingly, all these encodings make use of unsafe terms in some place. This suggests that such encoding is impossible in the safe lambda calculus and that the beta-equivalence problem may be simpler when restricted to safe terms. We have not been able to establish an upper-bound on the complexity of this problem. A lower-bound can however be obtained: the True Quantifier Boolean Formula (TQBF) problem (*i.e.*, deciding whether a quantified boolean formula is true) can be encoded in the safe lambda-calculus. Since the latter problem is PSPACE-complete, this implies that beta-equivalence for safe lambda terms is PSPACE-hard. A particularity of this encoding is that it relies on the entire type hierarchy and thus we only have PSPACE-hardness for the safe lambda calculus in its entirety. This contrasts with another result by Statman: there exists a finite set of types such that the beta-eta equivalence problem restricted to simply typed terms of these types is PSPACE-hard.

Extensions

PCF is the simply typed lambda calculus augmented with basic arithmetic operators, if-then-else branching and a family of recursion combinator $Y_A : ((A, A), A)$ for any type A . We define *safe* PCF to be PCF where the application and abstraction rules are constrained in the same way as the safe lambda calculus. This language inherits the good properties of the safe lambda calculus: No variable capture occurs when performing substitution and safety is preserved by the reduction rules of the small-step semantics of PCF. Similarly, we define *safe* IA as *safe* PCF augmented with the imperative features of Idealized Algol (IA for short) [Rey81]. A version of the no variable

capture lemma also holds in safe IA.

A concrete game semantics

Game semantics has emerged as a powerful paradigm for the study of higher-order functional programming languages in general, and in particular for the mother of all functional languages: the lambda calculus. The game approach was for instance the first to give rise to a full abstract model of PCF [AMJ94, HO00]. An inevitable question to ask is whether the safety constraint has a noticeable impact on the game denotation of a term. Answering this question would help us to gain a better understanding of the nature of the safety restriction on a fundamental level.

In the traditional presentation of game semantics, attention is taken to abstract away entirely the syntax of the language from the definition of the semantics. This syntax-independent aspect of game models constitutes their salient feature. But when it comes to analyze the safety restriction this is more a complication than a benefit because safety is precisely a *syntactic* constraint.

A substantial part of the thesis is therefore devoted to giving a presentation of game semantics that is more concrete than the traditional one in the sense that the semantic denotation of a term carries some information about its syntax. This presentation is based on ideas recently introduced by Ong [Ong06a]: a term is canonically represented by a certain abstract syntax tree of its η -long normal form referred as the *computation tree*. The computation itself is then described by a justified sequence of nodes of the computation tree respecting some formation rules and called a *traversal*. Essentially, traversals allow us to model β -reductions without altering the structure of the computation tree via substitution. A notable property is that *P-views* (in the game-semantic sense) of traversals corresponds to paths in the computation tree. We show that traversals are just representations of the *revealed game semantic* denotation (the set of uncovering of plays of the game-semantic denotation with respect to the syntax of the eta-long normal form). The standard game denotation can then be recovered by mean of a *reduction* operation which eliminates traversal nodes that are “internal” to the computation, thus implementing the counterpart of the hiding operation of game semantics. This leads to an isomorphism between the standard strategy denotation of a term and the set of reductions of traversals of its computation tree. Another contribution, on a more applied side, is the implementation of a tool to illustrate the theory of traversals and its correspondence with game semantics [Blu08].

We then extend our presentation of game semantics to PCF and IA. We accommodate the notion of computation tree to recursively defined terms as follows: the computation tree of a PCF term is defined as the least upper-bound of the chain of computation trees of its *syntactic approximants* [AM98b]. Think of it as the tree obtained by expanding Y-combinators *ad infinitum*. For instance the computation tree of $Y(\lambda f x. f x)$ is given by the abstract syntax tree of the η -long form of the infinite lambda-term $(\lambda f x. f x)((\lambda f x. f x)((\lambda f x. f x)(\dots))$. It is possible to define traversal rules modeling the arithmetic constants of PCF, so that a version of the Correspondence Theorem for PCF holds.

The extension to IA is complicated by the presence of the base type **var** used for reference variables. Indeed, the game denotation of **var** has infinitely many initial moves, therefore there is a mismatch between the tree representation of a term of type **var** and the arena underlying the game induced by the type **var**. It is also possible, however, to adapt the game-semantic correspondence to IA by replacing computation tree by computation hyper-trees. These are trees in which several nodes can be grouped into a single hyper-node.

This contribution constitutes a significant detour from the main topic of this thesis, but it is particularly useful to the analysis of the safety constraint.

Game semantics of safety

Using the correspondence result relating the game semantics of a lambda-term M to a set of *traversals* over its *computation tree*, we are able to show that safe terms are denoted by *P-incrementally justified strategies*. In such a strategy, pointers emanating from the P-moves of a play are uniquely reconstructible from the underlying sequence of moves and the pointers associated to the O-moves

therein: Specifically, a P-question always points to the last pending O-question in the P-view of greater order. (Consequently pointers in the game semantics of safe lambda-terms are only necessary from order 4 onwards, and for O-questions only.) More precisely, we show that a β -normal lambda-term is *safe* if and only if its strategy denotation is *P-incrementally justified*.

Model of safe lambda calculi

Our last contribution is to establish a game model of the safe lambda calculus. A fundamental result in theoretical computer science is the connection between Cartesian Closed Categories (CCC) and models of typed lambda calculi: it was observed by Lambek [Lam86] that any extensional model of the simply typed lambda calculus is a CCC, and conversely, any typed lambda calculus induces a CCC.

A similar categorical connection can be made for models of the safe lambda calculus. The categorical counterparts of safe lambda calculi are the *Incremental Closed Categories* (ICC). These categories are subcategories of CCC in which *Currying* is restrained. By showing that P-incrementally justified strategies compose, we can construct an ICC of games with morphisms given by P-incrementally justified strategies. This gives rise to a categorical game model of the safe lambda calculus.

Full abstraction

A common concept in game semantics is that the pure functional core of a programming language can be modeled by strategies verifying the properties of *visibility*, *innocence* and *well-bracketing*. Adding features to the language corresponds to relaxing one of these properties in the game model. For instance adding imperative features breaks innocence, adding exceptions-handling breaks well-bracketing and adding general references break visibility. Furthermore in each of these cases, the game model gives rise to a fully abstract model of the considered language. For instance the well-bracketed and visible strategies gives rise to a fully abstract game model of the language Idealized Algol (IA) (PCF extended with block-allocated variables).

Conversely, restricting the language corresponds to imposing more constraint on the strategy. As mentioned before, the strategy counterpart of the safety restriction is P-incremental justification. As expected, this restriction gives rise to a fully-abstract model of the safe fragment of PCF. These results are summarized in the following table:

Language	Strategy constraints
Safe IA	deterministic + visible + w.b. + P-i.j.
Safe PCF	deterministic + visible + w.b. + innocent + P-i.j.
PCF	deterministic + visible + w.b. + innocent
IA	deterministic + visible + w.b.
IA + exceptions	deterministic + visible
IA + exceptions + general references	deterministic

Algorithmic game semantics

The game semantic approach has become a very successful paradigm after solving the long-standing full abstraction problem of PCF; its success story did not stop here however. Game semantics turned out to be useful to the study of the observational equivalence problem: Given two terms, can they be used interchangeably? The research activity consisting of studying the observational equivalence problem via game semantics is called *Algorithmic game semantics*. A major breakthrough was the observation that the game model of Idealized Algol is effectively presentable [AM97] (a property that is not enjoyed by any model of PCF [Loa01]). This result paved the way to interesting characterization of the game denotation of lower-order IA terms. Ghica and McCusker observed [GM00] that pointers are unnecessary for representing plays in the game semantics of the second-order finitary fragment of Idealized Algol (IA_2 for short). Consequently observational equivalence for this fragment can be reduced to the problem of equivalence of regular

expressions. Similar characterizations were later obtained for other finitary fragments. For instance at order 3, although pointers are necessary, deciding observational equivalence of IA_3 is EXPTIME-complete [Ong04, MW05]. These results are all based on the same observation: at lower orders, the justification pointers present in the game denotation are either not required (*e.g.*, at order 2) or can be encoded succinctly (*e.g.*, at order 3). The possibility of representing plays *without some or all of their pointers* under the safety assumption strongly suggests that similar results can be obtained for safe fragments of IA.

Our last contribution consists therefore in studying the safety from the point of view of algorithmic game semantics. We introduce a new notion of observational equivalence for IA: A *safe context* is a safe IA term-in-context with a hole (a distinguished variable exactly once in the term); two terms are considered equivalent if no safe context can distinguish them. We show that up to order 3 this notion of observational equivalence coincides with the usual one. A basic result in Algorithmic game semantics is the Characterization Theorem: observational equivalence of two IA terms is characterized by the equality of their set of complete plays. We show a version of this theorem for our notion of observational equivalence: two terms are equivalent with respect to safe context if and only if they have the same set of P-incremental justified complete plays. Finally, based on these results, we show that all the known results about the complexity of observational equivalence up to order 3 are also true for our new notion of observational equivalence. The problem is known to be undecidable for the order-4 finitary fragment; We conjecture that observational equivalence for *safe terms* and with respect to *safe context* is decidable.

Prerequisite

The reader is assumed to be familiar with the simply-typed lambda calculus. A brief account is given in the background chapter; For more details, we refer the reader to introductions on the subject by Barendregt [Bar92] and Hindley [Hin97]. Familiarity with game semantics would be very helpful, in particular for Chapter 5 and 6, although it is not a requirement since the background chapter contains an introduction on the topic. A very good tutorial is [AM98b].

Organization of the thesis

The first chapter lays down the background for the rest of the thesis. It introduces briefly the simply-type lambda calculus and two of its extensions that will be studied throughout the thesis, namely PCF and Idealized Algol. It then presents *higher-order grammars*, the original setting in which the safety restriction firstly appeared, and presents the safety restriction with some related results. Finally, the last section is devoted to the presentation of the basics and main results of game semantics. It also fixes notation that will be used in other chapters when we study the game semantics of the safety restriction.

Chapter 3 introduces the definition of the *safe lambda calculus*. It establishes basic properties of the calculus and gives an account of its expressivity and complexity. The chapter concludes with a generalization of the safety restriction to other applied lambda calculi such as PCF and Idealized Algol.

Chapter 4 takes a detour from the safety restriction. It presents and extends the theory of traversals originally introduced by Ong [Ong06a]. It defines the notions of computation tree of a simply typed term and traversals over the computation tree. The ultimate goal of this chapter is to prove an important result called the Correspondence Theorem which establishes a correspondence between traversals of the computation tree and the game semantics of a term.

This correspondence theorem allows us to give in Chapter 5 an account of the game semantics of safety using a very simple syntactic argument.

Chapter 6 establishes a categorical model of the safe lambda calculus, safe PCF and safe Idealized Algol. A complete fully abstract game model is established. The chapter concludes with application to Algorithmic Game Semantics.

Chapter 2

Background

This chapter introduces in three sections the basic concepts that will be used throughout the thesis. The first section presents the lambda calculus, the second gives a brief introduction to higher-order grammars and present the original definition of the safety restriction. The last section is a condensed account of game semantics.

2.1 Lambda Calculus

We assume that the reader is familiar with the simply typed lambda calculus, but for precision and to fix notations we give here a brief overview of the basic definitions. For a detailed account on the subject, the reader is referred to the standard textbooks on the subject [Hin97, HS86, Bar92].

2.1.1 Terms

We fix a countable set of variables \mathcal{V} .

Definition 2.1.1. The set Λ of *terms* of the *untyped lambda calculus* is given by the set of derivations of the following grammar:

$$\Lambda = \mathcal{V} \mid \Lambda \Lambda \mid \lambda \mathcal{V} . \Lambda$$

These three basic formation rules are used to construct terms that are respectively *variables*, *applications* and *lambda-abstractions*.

A term is represented by an expression representing its derivation tree. It is computed as follows: the leaves of the derivation tree are concatenated from left to right and additional parentheses are added to indicate the order of the derivation. Parentheses ensure that the representation is unique. For instance they allow us to distinguish the five different derivations whose underlying concatenation of leaves is given by “ $\lambda x.MNQ$ ”; these derivations are $\lambda x.((MN)Q)$, $\lambda x.(M(NQ))$, $(\lambda x.M)(NQ)$, $(\lambda x.(MN))Q$, and $((\lambda x.M)N)Q$. We further use the following conventions:

- (i) we use symbols x, y, \dots to denote variables in \mathcal{V} and M, N, \dots to denote other terms;
- (ii) application associate to the left: MNQ stands for the term $((MN)Q)$;
- (iii) nested lambda abstractions are combined into a single one: $\lambda xyz.x$ stands for $\lambda x.\lambda y.\lambda z.x$. Also if \bar{x} denotes a sequence of variables $x_1 \dots x_n$ then we write $\lambda \bar{x}.M$ as a short-hand for $\lambda x_1 \dots \lambda x_n.M$.

Example 2.1.1. $\lambda x.x$, $\lambda x.xy$, $(\lambda x.xx)(\lambda x.xx)$ are all valid terms.

Definition 2.1.2. The set of **free variables** $FV(M)$ of a term M is given inductively by:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

An *occurrence* of a variable x in M is said to be **free** if it belongs to $FV(M)$. Otherwise it is said to be **bound**. A term M is **closed** if it has no free variable (*i.e.*, $FV(M) = \emptyset$).

We write $\text{closure}(M)$ to denote the closed term obtained from M by abstracting all its free variables by order of appearance in the term.

A variable is **fresh** if it does not occur anywhere in the terms that we are considering. Two terms M and N are **α -convertible** if one can be obtained from the other by renaming bound variables to fresh names. We consider syntactic equality of terms modulo α -conversion and we write $M \equiv N$ to denote this equality.

The set $\text{sub}(M)$ of **sub-terms** of M is given by induction as:

$$\begin{aligned} \text{sub}(x) &= \{x\} \\ \text{sub}(MN) &= \{MN\} \cup \text{sub}(M) \cup \text{sub}(N) \\ \text{sub}(\lambda x.M) &= \{\lambda x.M\} \cup \text{sub}(M) \end{aligned}$$

2.1.2 Substitution

Substitution refers to the transformation that replaces a free variable in a term by another term. The naive way to implement substitution consists in textually replacing all free occurrences of x in M by N . This is called *capture-permitting substitution*:

Definition 2.1.3. The **capture-permitting substitution** of N for x in M , written $M\{N/x\}$, is defined by induction as follows:

$$\begin{aligned} x_i\{N/x\} &\equiv N_i \\ y\{N/x\} &\equiv y \quad \text{if } x \neq y, \\ (M_1M_2)\{N/x\} &\equiv (M_1\{N/x\})(M_2\{N/x\}) \\ (\lambda x.M)\{N/x\} &\equiv \lambda x.M \\ (\lambda y.M)\{N/x\} &\equiv \lambda y.M\{N/x\} \text{ if } y \neq x. \end{aligned}$$

Although this definition is valid, it is not adequate in the sense that it is not sound with respect to syntactical equality: take the terms $M_1 \equiv \lambda y.x$, $M_2 \equiv \lambda z.x$ and $N \equiv y$. We have $M_1\{N/x\} \equiv \lambda y.y$ and $M_2\{N/x\} \equiv \lambda z.$ Thus although M_1 and M_2 are syntactically equivalent, performing the same substitution on both terms gives terms that are not syntactically equivalent.

The source of the problem lies the last equation: in the abstraction case, when pushing the substitution under the lambda, some care needs to be taken so that the free-variables in M do not get “captured” by the abstraction. This is traditionally achieved by renaming all the free variables in M afresh before continuing with the substitution:

Definition 2.1.4. The **substitution** of N for x in M written $M[N/x]$ is defined by induction as follows:

$$\begin{aligned} x[t/x] &\equiv t \\ y[t/x] &\equiv y \quad \text{if } x \neq y, \\ (M_1M_2)[t/x] &\equiv (M_1[t/x])(M_2[t/x]) \\ (\lambda x.M)[t/x] &\equiv \lambda x.M \\ (\lambda y.M)[t/x] &\equiv \lambda z.M[z/y][t/x] \text{ if } x \neq y \text{ and where } z \text{ is a fresh variable.} \end{aligned}$$

Observe that only the last equation differs from the previous definition.

These two operations can be straightforwardly generalized to many variables. This is called *simultaneous substitution*:

Definition 2.1.5. The *simultaneous capture-permitting substitution* of N_1, \dots, N_n for the (distinct) variables x_1, \dots, x_n in M , written $M\{N_1/x_1, \dots, N_n/x_n\}$ and abbreviated here as $M\{\overline{N}/\overline{x}\}$ is defined by induction as follows:

$$\begin{aligned} x_i \{\overline{N}/\overline{x}\} &\equiv N_i \\ y \{\overline{N}/\overline{x}\} &\equiv y \quad \text{if } y \neq x_i \text{ for all } i, \\ (M_1 M_2) \{\overline{N}/\overline{x}\} &\equiv (M_1 \{\overline{N}/\overline{x}\})(M_2 \{\overline{N}/\overline{x}\}) \\ (\lambda x_i. M) \{\overline{N}/\overline{x}\} &\equiv \lambda x_i. M \{N_1 \dots N_{i-1} N_{i+1} \dots N_n / x_1 \dots x_{i-1} x_{i+1} \dots x_n\} \\ (\lambda y. M) \{\overline{N}/\overline{x}\} &\equiv \lambda y. M \{\overline{N}/\overline{x}\} \text{ if } y \neq x_i \text{ for all } i. \end{aligned}$$

Definition 2.1.6. The *simultaneous substitution* of N_1, \dots, N_n for the (distinct) variables x_1, \dots, x_n in M , written $M[N_1/x_1, \dots, N_n/x_n]$ and abbreviated here as $M[\overline{N}/\overline{x}]$ is defined by induction as follows:

$$\begin{aligned} x_i [\overline{N}/\overline{x}] &\equiv N_i \\ y [\overline{N}/\overline{x}] &\equiv y \quad \text{if } y \neq x_i \text{ for all } i, \\ (MN) [\overline{N}/\overline{x}] &\equiv (M [\overline{N}/\overline{x}])(N [\overline{N}/\overline{x}]) \\ (\lambda x_i. M) [\overline{N}/\overline{x}] &\equiv \lambda x_i. M [N_1 \dots N_{i-1} N_{i+1} \dots N_n / x_1 \dots x_{i-1} x_{i+1} \dots x_n] \\ (\lambda y. M) [\overline{N}/\overline{x}] &\equiv \lambda z. M [z/y] [\overline{N}/\overline{x}] \\ &\quad \text{if } y \neq x_i \text{ for all } i \text{ and where } z \text{ is a fresh variable.} \end{aligned}$$

2.1.3 Conversion

A binary relation R over Λ is **compatible** if $M R M'$ implies $MN \rightarrow_\beta M'N$, $NM \rightarrow_\beta NM'$ and $\lambda x. M \rightarrow_\beta \lambda x. M'$ for all $M, M', N \in \Lambda$. It is **R transitive** if $M \rightarrow_\beta N$ and $N \rightarrow_\beta Q$ implies $M \rightarrow_\beta Q$; **reflexive** if $M \rightarrow_\beta M$; and **symmetric** if $M \rightarrow_\beta N$ implies $N \rightarrow_\beta M$, for all $M, N, Q \in \Lambda$.

The concept of computation in the lambda calculus is incarnated by a rewriting rule for terms called **β -reduction**:

Definition 2.1.7. We call **β -redex** any term of the form $(\lambda x. M)N$. It *contraction* is defined as $M[N/x]$. We define β as the relation mapping a redex to its contraction:

$$\beta = \{((\lambda x. M)N, M[N/x]) \mid M, N \in \Delta, x \in \mathcal{V}\}.$$

The **one-step β -reduction** relation \rightarrow_β is defined as the compatible closure of the relation β .

The relation \rightarrow_β denotes the reflexive transitive closure of \rightarrow_β , and the relation $=_\beta$, called **β -equality** or also **β -conversion**, denotes the reflexive symmetric transitive closure of \rightarrow_β .

In addition to the β -reduction rule the **η -reduction** \rightarrow_η is defined as the smallest compatible relation verifying:

$$\lambda z. Mz \rightarrow_\eta M \quad \text{if } z \notin FV(M).$$

We define η -conversion $=_\eta$ as the reflexive symmetric transitive closure of \rightarrow_η .

Definition 2.1.8 (Normal form). A term

- (i) is a **β -normal form**, β -nf for short, if it does not contain any β -redex.
- (ii) *has* a β -normal form, or is **normalizable**, if it is β -equal to a β -normal form.
- (iii) is **strongly normalizable** if every sequence of reduction starting from it is finite (and therefore ends with a normal form).

The notions of η and $\beta\eta$ normal form are defined similarly.

2.1.4 Properties

A reduction is *weakly normalizing* if every term is normalizable and *strongly normalizing* if every term is strongly normalizable. The (untyped) lambda calculus is not even weakly normalizing with respect to β -reduction since for instance the term $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ β -reduces to itself.

The lambda calculus verified the so-called *Church-Rosser* theorem:

Theorem 2.1.1 (Church-Rosser Theorem). *If $M \rightarrow_\beta N_1$ and $M \rightarrow_\beta N_2$ then for some N we have $N_1 \rightarrow_\beta N$ and $N_2 \rightarrow_\beta N$.*

This is sometimes summarized as “ \rightarrow_β verifies the diamond property”. A consequence of this theorem is that a term has at most one β -normal form. Furthermore:

Theorem 2.1.2 (Normalization Theorem [Bar84]). *The leftmost reduction strategy is normalizing (i.e., if M has a normal form then the reduction strategy consisting in contracting the leftmost redex leads to that normal form).*

2.1.5 Simple types

Simple types are objects that are constructed from atomic types using the function space arrow operator \rightarrow . Formally, we fix a set \mathbb{A} of *atomic types* and we define the set $\mathbb{T}_{\mathbb{A}}$ of *simple types* over \mathbb{A} as the set generated from the following grammar:

$$\mathbb{T}_{\mathbb{A}} ::= \mathbb{A} \mid \mathbb{T}_{\mathbb{A}} \rightarrow \mathbb{T}_{\mathbb{A}}$$



We will use the greek letter symbols α, β, \dots to refer to atomic types and capital letters A, B, \dots to refer to other types. We further assume that \mathbb{A} has a distinguished atomic type denoted by the symbol o .

By convention, \rightarrow associates to the right. Thus every type can be written as $A_1 \rightarrow \dots \rightarrow A_n \rightarrow \alpha$ for some atomic type α , which we shall abbreviate to $(A_1, \dots, A_n, \alpha)$ (in case $n = 0$, we identify (α) with α). The number n is called the *arity* of the type, it is written $\text{arity}(T)$ for any type T .

CONVENTION 2.1.1 We use the following abbreviations for types:

- (i) For any atom a and natural number $n \in \mathbb{N}$, we define the types n_a as follows: $0_a = a$ and $(n+1)_a = n_a \rightarrow a$;

- (ii) For any types A, B and positive natural number $n > 0$, the type $A^n \rightarrow B$ is defined by induction as: $A^1 \rightarrow B = A \rightarrow B$ and $A^{n+1} \rightarrow B = A \rightarrow (A^n \rightarrow B)$. In other words:

$$A^n \rightarrow B = \overbrace{A \rightarrow \dots \rightarrow A}^{n \text{ times}} \rightarrow B;$$

- (iii) For any types A, B and atomic type α we write $A[B/\alpha]$ to denote the type obtained from A by substituting occurrences of α by B . Similarly for any context Γ we write $\Gamma[B/\alpha]$ to denote the context obtained by performing the substitution on each type of the context.

The *order* of a type is given by $\text{ord } \alpha = 0$ for any atomic type α and $\text{ord } (A \rightarrow B) = \max(1 + \text{ord } A, \text{ord } B)$. We assume an infinite set of typed variables. The order of a typed term or symbol is defined to be the order of its type.

2.1.6 Simply typed lambda calculus à la Curry

There exist two styles of presentation of the simply typed lambda calculus. In the Curry style, typing is implicit. This means that each untyped term is assigned either no type or infinitely many types. The other presentation, called Church style, makes the typing information explicit in the structure of the term by introducing type annotations in it. Thus terms of this system have a unique type. We present here the Curry version of the simply typed lambda calculus.

We write $M : A$ to denote that the term M can be assigned the type $A \in \mathbb{T}$ in the typing-system. A set Γ of *typing assumptions* is a set of typing-assignments of the form $x : T$ where x is a variable in \mathcal{V} and $T \in \mathbb{T}$. It is *consistent* if all the the variables names are distinct (*i.e.*, each variable name is assigned a unique type). The underlying set of variable names is called the domain Γ and is written $\text{dom}(\Gamma)$. We will write $\Gamma, x : A$ to denote the set of typing assumptions $\Gamma \cup \{x : A\}$. We consider judgments of the form $\Gamma \vdash_{\text{Cu}} M : A$ called **terms-in-context** where Γ is a *consistent* set of typing assumptions called the **typing context**, A is a simple type and M is a term.

Definition 2.1.9. The **simply typed lambda calculus à la Curry**, denoted by $\Lambda_{\rightarrow}^{\text{Cu}}$, is defined as the set of terms-in-context of the form $\Gamma \vdash_{\text{Cu}} M : A$ that are derivable from the variable, application and abstraction rules defined as follows:

$$\frac{}{\Gamma \vdash_{\text{Cu}} x : A} \quad x : A \in \Gamma \qquad \frac{\Gamma \vdash_{\text{Cu}} M : A \rightarrow B \quad \Gamma \vdash_{\text{Cu}} N : A}{\Gamma \vdash_{\text{Cu}} MN : B} \qquad \frac{\Gamma, x : A \vdash_{\text{Cu}} M : B}{\Gamma \vdash_{\text{Cu}} \lambda x.M : A \rightarrow B}$$

Whenever the context is empty we just write $\vdash_{\text{Cu}} M : A$ instead of $\emptyset \vdash_{\text{Cu}} M : A$.

In the literature, the second and third rules are sometimes called the \rightarrow -elimination and \rightarrow -introduction rules respectively.

The notion of “derivability” used in the above definition can be made more precise: A *typing derivation* or **typing deduction** Δ of $\Lambda_{\rightarrow}^{\text{Cu}}$ is a tree labelled by terms-in-context of the form $\Gamma \vdash_{\text{Cu}} M : A$ where the leaves are axioms and the internal nodes are deduced from their children nodes using the rules of $\Lambda_{\rightarrow}^{\text{Cu}}$. The edges of the tree also have labels indicating the rule used to make the deductions. The root of the tree is called the *conclusion* of the derivation. Such tree is usually represented with leaves at the top and root at the bottom [Hin97]. Terms-in-context of the simply typed lambda calculus are then defined as the set of conclusions of derivations in $\Lambda_{\rightarrow}^{\text{Cu}}$.

An **inhabitant** of a type $T \in \mathbb{T}$ is a term $M \in \Lambda$ such that for some typing-context Γ we have $\Gamma \vdash_{\text{Cu}} M : T$.

We now recall some standard results:

Property 2.1.1 (Weakening). Suppose $\Gamma \vdash_{\text{Cu}} M : A$ and Γ' is a typing-context with $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash_{\text{Cu}} M : A$.

Proposition 2.1.1 (Typability of subterms). *Let M' be a subterm of M . Then if $\Gamma \vdash_{\text{Cu}} M : A$ then $\Gamma' \vdash_{\text{Cu}} M' : A'$ for some context Γ' and type A' .*

Lemma 2.1.1 (Substitution Lemma).

- (i) If $\Gamma, x : A \vdash_{\text{Cu}} M : B$ and $\Gamma \vdash_{\text{Cu}} N : A$ then $\Gamma \vdash_{\text{Cu}} M [N/x] : B$.
- (ii) If $\Gamma \vdash_{\text{Cu}} M : A$ then $\Gamma [B/\alpha] \vdash_{\text{Cu}} N : A [B/\alpha]$.

Theorem 2.1.3 (Subject Reduction). *Suppose that $M \rightarrow_{\beta} N$. Then*

$$\Gamma \vdash_{\text{Cu}} M : A \implies \Gamma \vdash_{\text{Cu}} M' : A \text{ .}$$

2.1.6.1 Typing problems

The three following problems are often considered in the lambda calculus:

- **TYPE CHECKING:** Given a term M , context Γ and type A , do we have $\Gamma \vdash_{\text{Cu}} M : A$?
- **TYPABILITY:** Given a term M and context Γ , is there a type A such that $\Gamma \vdash_{\text{Cu}} M : A$?
- **INHABITATION:** Given a type A , is there a term M such that $\vdash_{\text{Cu}} M : A$?

Definition 2.1.10 (Type substitution). A **type substitution** is an expression $[T_1/a_1, \dots, T_n/a_n]$ where a_1, \dots, a_n are distinct atomic types in \mathbb{A} and $T_1, \dots, T_n \in \mathbb{T}$.

For any type $T \in \mathbb{T}$ and type substitution $[T_1/a_1, \dots, T_n/a_n]$ we define $T[T_1/a_1, \dots, T_n/a_n]$ to be the type obtained from T by substituting T_1 for a_1, \dots, T_n for a_n . The resulting type is called an **instance** of the type T .

The operation of substitution naturally extends to finite sequences of types, contexts, terms-in-context and deductions.

Definition 2.1.11 (Principality). A term M has **principal type** A if for every possible derivation $\vdash_{\text{Cu}} M : A'$, A' is an instance of A . A **principal deduction** for a term M is a deduction Δ of the term-in-context $\Gamma \vdash_{\text{Cu}} M : T$ such that every other deduction with the same conclusion is an instance of Δ , so in particular T is a **principal type** of M .

Theorem 2.1.4 (PT Theorem, Curry, Hindley, Milner)). *It is decidable whether a term is typable in Λ_{\rightarrow} . Moreover if M is typable then it has a **principal deduction** that is computable from M .*

This implies that both TYPE CHECKING and TYPABILITY are decidable.

Theorem 2.1.5 (Strong normalization, Tait [Tai67]). *Every term that is typable in Λ_{\rightarrow} is strongly normalizable (i.e., every reduction sequence leads to its (unique) normal form).*

Theorem 2.1.6 (Statman [Sta79a]). *The problem INHABITATION for types defined over an infinite number of atoms is PSPACE-complete (and thus decidable).*

2.1.7 Simply typed lambda calculus à la Church

The simply typed lambda calculus that we have introduced corresponds to the *Curry-style* version. There is another approach called the *Church-style* presentation in which variable binders are annotated with types¹. The set of annotated-types $\Lambda_{\mathbb{T}}$ is formally given by the following grammar:

$$\Lambda_{\mathbb{T}} = \mathcal{V} \mid \Lambda_{\mathbb{T}} \Lambda_{\mathbb{T}} \mid \lambda_{\mathbb{T}} \mathcal{V} : \mathbb{T}. \Lambda_{\mathbb{T}}$$

Observe that in the abstraction case, the binder is annotated with a type. That is the only difference with untyped terms Λ . For any annotated term $M \in \Lambda_{\mathbb{T}}$, the untyped term underlying M , written $|M|$, is obtained by erasing all the type annotations from M .

We can now introduce new judgments of the form

$$\Gamma \vdash_{\text{Ch}} M : A \in \Gamma$$

where M ranges over *annotated terms* $\Lambda_{\mathbb{T}}$. The simply typed lambda calculus à la Church, written $\Lambda_{\rightarrow}^{\text{Ch}}$, is then given by the following typing system:

$$\frac{}{\Gamma \vdash_{\text{Ch}} x : A} x : A \in \Gamma \quad \frac{\Gamma \vdash_{\text{Ch}} M : A \rightarrow B \quad \Gamma \vdash_{\text{Ch}} N : A}{\Gamma \vdash_{\text{Ch}} MN : B} \quad \frac{\Gamma, x : A \vdash_{\text{Ch}} M : B}{\Gamma \vdash_{\text{Ch}} \lambda x^A. M : A \rightarrow B}$$

In contrast with the Curry version, terms of the Church typed lambda calculus have a unique type at most:

Proposition 2.1.2 (Uniqueness of types in $\Lambda_{\rightarrow}^{\text{Ch}}$). *If $\Gamma \vdash_{\text{Ch}} M : T$ and $\Gamma \vdash_{\text{Ch}} M : T'$ then $T = T'$. Further if $\Gamma \vdash_{\text{Ch}} M : T$, $\Gamma \vdash_{\text{Ch}} M' : T'$ and $M =_{\beta} M'$ then $T = T'$.*

The Curry-style and Church-style systems are related by the following result:

Proposition 2.1.3. (i) *Let $M \in \Lambda_{\mathbb{T}}$. Then $\Gamma \vdash_{\text{Ch}} M : A \implies \Gamma \vdash_{\text{Cu}} |M| : A$.*

(ii) *Let $M \in \Lambda$. Then $\Gamma \vdash_{\text{Cu}} M : A \implies \exists M' \in \Lambda_{\mathbb{T}} \text{ s.t. } \Gamma \vdash_{\text{Ch}} M' : A \wedge |M'| = M$.*

¹In fact in the original Church presentation, variable occurrences are also annotated. The version that we present here is sometimes called the Bruijn-style simply typed lambda calculus. These two presentations are essentially equivalent.

In particular this implies

Corollary 2.1.7. *Let $T \in \mathbb{T}$. Then T is inhabited in $\Lambda_{\rightarrow}^{\text{Ch}}$ iff it is inhabited in $\Lambda_{\rightarrow}^{\text{Cu}}$.*

CONVENTION 2.1.2 In the rest of this thesis we will use judgments of the form $\Gamma \vdash_{\text{st}} M : A$ to denote both *à la* Curry and *à la* Church terms-in-context: if M is an annotated term in $\Lambda_{\mathbb{T}}$ then the judgment stands to $\Gamma \vdash_{\text{Ch}} M : A$ whereas if M is an untyped term in Λ then it stands for $\Gamma \vdash_{\text{Cu}} M : A$.

2.1.8 Extensions

The simply typed lambda calculus can be extended with a set of typed constants Ξ . To allow the use of constants, the syntax of Λ is modified with a new grammar rule: $\Lambda = \dots \mid \Xi \mid \dots$. The typing system is then augmented with the rule

$$(\text{const}) \frac{}{\vdash_{\text{Cu}} f : A} f \in \Xi.$$

A new notion of reduction is defined to allow contraction of terms whose head occurrence is a Ξ -constant: Every constant c in Ξ comes with a rewriting function $f_c : \Lambda^k \rightarrow \Lambda$ for some $k \in \mathbb{N}$ determining the interpretation of the constant. The following rule is then added to those of the lambda calculus: $cM_1 \dots M_k \rightarrow f_c(M_1, \dots, M_k)$ for every closed normal forms M_1, \dots, M_k .

2.1.9 PCF

The *Programming language for Computable Functions*, PCF for short, is a simple programming language based on the *Logic of Computable Functions* (LCF) devised by Dana Scott [Sco69]. It was introduced in a classical paper by Plotkin “LCF considered as a programming language” [Pl77]. PCF can be viewed as the Church-like simply typed lambda calculus extended with arithmetic operators, conditional and recursion.

Syntax

The set of types is \mathbb{T}_{exp} : the simple types generated from the atomic type **exp** representing natural numbers. PCF terms are given by the grammar:

$$\begin{aligned} M ::= & x \mid \lambda x^A. M \mid MM \mid \\ & \mid n \mid \text{succ } M \mid \text{pred } M \\ & \mid \text{cond } MMM \mid Y_A M \end{aligned}$$

where x ranges over a set of countably many variables, n represents an integer constant ranging over the set of natural numbers, **succ** represents the successor function on integer, **pred** the predecessor, **cond** the conditional (*i.e.*, if-then-else branching) and $Y_A : (A \rightarrow A) \rightarrow A$ for any type A is the recursion combinator.

The language is formally given by terms-in-context of the form $\Gamma \vdash M : A$ defined by induction over the rules of Table 2.1.

Example 2.1.2. The integer addition function is definable in PCF by:

$$\text{PLUS} \equiv Y(\lambda p x y. \text{cond } x y (p (\text{pred } x) (\text{succ } y)))$$

so that for terms M and N , if $M \Downarrow m$ and $N \Downarrow n$, $m, n \in \mathbb{N}$ then $\text{PLUS } M N \Downarrow m + n$.

Equality on integer is also definable by:

$$\begin{aligned} \text{EQ} = & Y(\lambda f^{\text{exp} \rightarrow \text{exp} \rightarrow \text{exp}} x^{\text{exp}} y^{\text{exp}}. \text{cond } a \\ & (\text{cond } b \ 1 \ 0) \\ & (\text{cond } b \ 0 \ (f (\text{pred } a) (\text{pred } b)))) \end{aligned}$$

so that $\text{EQ } M N \Downarrow 1$ if M and N evaluate to the same value, and $\text{EQ } M N \Downarrow 0$ otherwise.

$$\begin{array}{c}
(\text{var}) \frac{}{x_1 : A_1, x_2 : A_2, \dots x_n : A_n \vdash x_i : A_i} \quad i \in 1..n \\
(\text{app}) \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \quad (\text{abs}) \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \rightarrow B} \\
(\text{const}) \frac{}{\Gamma \vdash n : \text{exp}} \quad (\text{succ}) \frac{\Gamma \vdash M : \text{exp}}{\Gamma \vdash \text{succ } M : \text{exp}} \quad (\text{pred}) \frac{\Gamma \vdash M : \text{exp}}{\Gamma \vdash \text{pred } M : \text{exp}} \\
(\text{cond}) \frac{\Gamma \vdash M : \text{exp} \quad \Gamma \vdash N_1 : \text{exp} \quad \Gamma \vdash N_2 : \text{exp}}{\Gamma \vdash \text{cond } M N_1 N_2 : \text{exp}} \quad (\text{rec}) \frac{\Gamma \vdash M : A \rightarrow A}{\Gamma \vdash Y_A M : A}
\end{array}$$

Table 2.1: Formation rules for PCF terms.

Operational semantics

The operational semantics of the language is given using a big-step style semantics. We call **canonical form** a term that is either a number or a function:

$$V ::= n \mid \lambda x^A. M$$

The notation $M \Downarrow V$ means that the closed term M evaluates to the canonical form V . We write $M \Downarrow$ if the judgment $M \Downarrow V$ is valid for some canonical form V . The full operational semantics is given in Table 2.2. Since the evaluation rules are defined for closed terms only, the context Γ is omitted in the rules.

$$\begin{array}{c}
\frac{}{\overline{V} \Downarrow V} \quad \text{provided that } V \text{ is in canonical form.} \\
\frac{M \Downarrow \lambda x. M' \quad M' [x/N] \Downarrow V}{MN \Downarrow V} \\
\frac{M \Downarrow n}{\text{succ } M \Downarrow n+1} \quad \frac{M \Downarrow n+1}{\text{pred } M \Downarrow n} \quad \frac{M \Downarrow 0}{\text{pred } M \Downarrow 0} \\
\frac{M \Downarrow 0 \quad N_1 \Downarrow V}{\text{cond } MN_1N_2 \Downarrow V} \quad \frac{M \Downarrow n+1 \quad N_2 \Downarrow V}{\text{cond } MN_1N_2 \Downarrow V} \\
\frac{M(YM) \Downarrow V}{YM \Downarrow V}
\end{array}$$

Table 2.2: Big-step operational semantics of PCF.

Case constructs

PCF is sometimes extended with a family of k -ary conditionals formed with the rule:

$$(\text{case}) \frac{\Gamma \vdash M : \text{exp} \quad \Gamma \vdash N_1 : \text{exp} \quad \dots \quad \Gamma \vdash N_k : \text{exp}}{\Gamma \vdash \text{case}_k M N_1 N_2 \dots N_k : \text{exp}}$$

The resulting language is referred as PCF_c . Its operational semantics is given by that of PCF together with the rule:

$$\frac{M \Downarrow i \quad N_{i+1} \Downarrow V}{\text{case}_k N N_1 N_2 \dots N_k \Downarrow V} \quad i \in \{0, \dots, k-1\}.$$

Syntactic sugar

For any integer $k \in \mathbb{N}$ and term $M : \mathbf{exp}$ we write “ $M + k$ ” as syntactic sugar for “ $\mathbf{PLUS} M k$ ”. For any terms M, N_1 and N_2 of type \mathbf{exp} we write “ $N_1 = N_2$ ” for “ $\mathbf{EQ} N_1 N_2$ ”, “ $N_1 \neq N_2$ ” for “ $\mathbf{cond}(\mathbf{EQ} N_2 N_2) 1 0$ ”, and “ $\mathbf{if} M \text{ then } N_1 \text{ else } N_2$ ” for “ $\mathbf{cond} M N_2 N_1$ ”. We will also use the construct

$$\begin{array}{l} \mathbf{match} M \text{ with} \\ \quad a_1 \rightarrow N_1 \\ \quad | \dots \\ \quad | a_q \rightarrow N_q \\ \quad | _ \rightarrow R \end{array}$$

for distinct integers a_1, \dots, a_q , $q \geq 1$, as syntactic sugar for “ $\mathbf{case}_m M N'_1 \dots N'_m$ ” where $m = 1 + \max_{1 \leq i \leq q} a_i$ and for $1 \leq j \leq m$, $N'_j \equiv N_i$ if $j = a_i$ for some $1 \leq i \leq q$ and $N'_j \equiv R$ otherwise.

2.1.10 Idealized Algol

Idealized Algol, IA for short, is an extension of PCF with imperative features that was introduced by J.C. Reynold [Rey81]. It adds imperative features such as local variables and sequential composition. Its types is given by the simple types over the basis $\{\mathbf{com}, \mathbf{exp}, \mathbf{var}\}$ where \mathbf{com} denotes the type of commands and \mathbf{var} the type of local variables.

The most basic command is given by the constant \mathbf{skip} of type \mathbf{com} which performs no computation. Commands can be composed using the sequential composition operator \mathbf{seq}_A for any base type A . The sequential composition of two terms $N_0 : \mathbf{com}$ and $N_1 : A$ is given by the term $M = \mathbf{seq}_A N_0 N_1 : \mathbf{com}$ which is interpreted operationally as follows: N_0 is evaluated first and if it terminates then the term N_1 is evaluated. In the case where $A = \mathbf{exp}$, the result of the evaluation of N_1 is returned; otherwise $A = \mathbf{com}$ and the command N_1 is just evaluated after N_0 and the expression yields no result. Terms formed with the operator $\mathbf{seq}_{\mathbf{exp}}$ are called *active expressions*.

Local variables are declared using the \mathbf{new} operator, their content is modified using \mathbf{assign} and retrieved using \mathbf{deref} . Operationally, these variables behave like memory cells.

In addition to these local variables, IA features the so called “*bad variable construct*” \mathbf{mkvar} . This operator can be used to construct a special variable by specifying custom assignment and dereferencing functions. It takes two arguments: the first one, the acceptor, is the function that is responsible of affecting a value to the variable. The second one is an expression that returns the value hold by the variable. This mechanism is similar to the “set/get” object programming paradigm used by C++ programmers. An example of such variable is the term $\mathbf{mkvar} (\lambda v. \mathbf{skip}) 0$. Variables created that way are called “bad variables” because they do not necessarily behave like a memory cell: reading the content of the variable does not necessarily gives you the last value that was written. For instance reading from the variable defined above always yield 0 whichever value was written to it previously.

This addition to the language may seem a little bit artificial but its presence has semantic importance.²

The syntax

The typing system for IA is an extension of that of PCF. The additional rules are given in Table 2.3.

We will sometimes use the ML-like syntactic sugar: “ $X := v$ ” for “ $\mathbf{assign} X v$ ”, “ $!X$ ” for “ $\mathbf{deref} X$ ”, and “ $M; N$ ” for “ $\mathbf{seq} M N$ ”.

²McCusker showed that the standard game model of IA is only equationally fully abstract for the language without bad variables, whereas for full IA, it is also *inequationally* fully abstract [McC03].

$$\begin{array}{c}
\frac{\Gamma \vdash M : \text{com} \quad \Gamma \vdash N : A}{\Gamma \vdash \text{seq}_A M N : A} \quad A \in \{\text{com}, \text{exp}\} \\
\\
\frac{\Gamma \vdash M : \text{var} \quad \Gamma \vdash N : \text{exp}}{\Gamma \vdash \text{assign } M N : \text{com}} \quad \frac{\Gamma \vdash M : \text{var}}{\Gamma \vdash \text{deref } M : \text{exp}} \\
\\
\frac{\Gamma, x : \text{var} \vdash M : A}{\Gamma \vdash \text{new } x \text{ in } M} \quad A \in \{\text{com}, \text{exp}\} \\
\\
\frac{\Gamma \vdash M_1 : \text{exp} \rightarrow \text{com} \quad \Gamma \vdash M_2 : \text{exp}}{\Gamma \vdash \text{mkvar } M_1 M_2 : \text{var}}
\end{array}$$

Table 2.3: Formation rules for IA.

Finitary fragments of Idealized algol

We call **Finitary Idealized Algol** the recursion-free sub-fragment of IA defined over finite ground types (*i.e.*, the atomic type **exp** inhabits the set $\{0..M\}$ for some fixed natural number $M \in \mathbb{N}$).

Definition 2.1.12 (*i*th order IA term). A term $\Gamma \vdash M : T$ of finitary Idealized algol is an *i*th-order term if any sequent $\Gamma' \vdash N : A$ appearing in the typing derivation of M is such that $\text{ord } A \leq i$ and all the variables in Γ' are of order strictly less than *i*.

The fragment of finitary Idealized Algol consisting of the collection of *i*th-order terms is denoted IA_i and is called the **order-*i* finitary fragment of IA**. If we add the iteration construct defined as

$$\frac{\Gamma \vdash M : \text{bool} \quad \Gamma \vdash N : \text{com}}{\Gamma \vdash \text{while } M \text{ do } N : \text{com}} \quad \text{where } \forall x \in \Gamma : \text{ord } x < i$$

we obtain the fragments $IA_i + \text{while}$ for $i \in \mathbb{N}$. Finally $IA_i + Y_j$ for $j < i$ denotes the fragment IA_i augmented with a set of fixed-point iterators $Y_A : (A \rightarrow A) \rightarrow A$ for any type A of order j at most, whose syntax is defined by the rule:

$$\frac{\Gamma \vdash \lambda x^A. M : A \rightarrow A}{\Gamma \vdash Y_A M : A} \quad \text{where } \forall x \in \Gamma : \text{ord } x < i \text{ and } \text{ord } A \leq j.$$

Operational semantics of IA

To define the operational semantics of IA we proceed slightly differently than for PCF. Instead of giving the semantics for closed terms, we consider terms whose free variables are all of type **var**. A context Γ whose variables are all assigned the type **var** is called a **var-context**. Terms are “closed” by means of *stores*. A **store** is a function mapping free variables of type **var** to natural numbers. It is called Γ -store just if its domain of definition is precisely the domain of the typing-context Γ . If s is a store then $s \mid x \mapsto n$ denotes the store that maps x to n and acts according to s for other variables.

The set of IA **canonical forms** is given by the grammar:

$$V ::= \text{skip} \mid n \mid \lambda x^A. M \mid x \mid \text{mkvar } M N$$

where n ranges over natural number and x over variable names.

An IA **program** is a term together with a Γ -store such that $\Gamma \vdash M : A$. The evaluation semantics is expressed by the judgment form:

$$s, M \Downarrow s', V$$

where s and s' are Γ -stores, V is a canonical form and $\Gamma \vdash V : A$.

The operational semantics for IA is given by the rule of PCF (Table 2.2) together with the rules of Table 2.4 in which the following abbreviation is used:

$$\begin{array}{c}
 \frac{M_1 \Downarrow V_1 \quad M_2 \Downarrow V_2}{M \Downarrow V} \quad \text{for} \quad \frac{s, M_1 \Downarrow s', V_1 \quad s', M_2 \Downarrow s'', V_2}{s, M \Downarrow s'', V} \\
 \\
 \textbf{Sequencing:} \quad \frac{M \Downarrow \text{skip} \quad N \Downarrow V}{\text{seq } M \ N \Downarrow V} \\
 \\
 \textbf{Variables:} \quad \frac{s, N \Downarrow s', n \quad s', M \Downarrow s'', x}{s, \text{assign } M \ N \Downarrow (s'' \mid x \mapsto n), \text{skip}} \quad \frac{s, M \Downarrow s', x}{s, \text{deref } M \Downarrow s', s'(x)} \\
 \\
 \textbf{Bad-variables:} \quad \frac{N \Downarrow n \quad M \Downarrow \text{mkvar } M_1 \ M_2 \quad M_1 \ n \Downarrow \text{skip}}{\text{assign } M \ N \Downarrow \text{skip}} \quad \frac{N \Downarrow \text{mkvar } M_1 \ M_2 \quad M_2 \Downarrow n}{\text{deref } M \Downarrow n} \\
 \\
 \textbf{Block:} \quad \frac{(s \mid x \mapsto 0), M \Downarrow (s' \mid x \mapsto n), V}{s, \text{new } x \text{ in } M \Downarrow s', V}
 \end{array}$$

Table 2.4: Big-step operational semantics of IA.

Small-step semantics

The operational semantics of IA can equivalently be defined by means of a small-step semantics: We use reduction rules of the form $s, M \rightarrow s', M'$ where s and s' denotes the stores and M and M' denotes IA terms. The relation \rightarrow is defined by the following rules (We write $M \rightarrow M'$ as an abbreviation for $s, M \rightarrow s', M'$):

- *β -reduction:* If $M\beta M'$ then $M \rightarrow M'$;
- *PCF constants:*

$$\begin{array}{lcl}
 \text{succ } n & \rightarrow & n + 1 \\
 \text{pred } n + 1 & \rightarrow & n \\
 \text{pred } 0 & \rightarrow & 0 \\
 \text{cond } 0 \ N_1 N_2 & \rightarrow & N_1 \\
 \text{cond } (n + 1) \ N_1 N_2 & \rightarrow & N_2 \\
 Y \ M & \rightarrow & M(YM)
 \end{array}$$

- *IA constants:*

$$\begin{array}{lcl}
 \text{seq skip } M & \rightarrow & M \\
 s, \text{new } x \text{ in } M & \rightarrow & (s \mid x \mapsto 0), M \\
 s, \text{assign } x \ n & \rightarrow & (s \mid x \mapsto n), \text{skip} \\
 s, \text{deref } x & \rightarrow & s, s(x) \\
 \text{assign (mkvar } MN) \ n & \rightarrow & Mn \\
 \text{deref (mkvar } MN) & \rightarrow & N
 \end{array}$$

where n ranges over the natural numbers.

The *redexes*—the expressions occurring in the left-hand side of a reduction rule—can be reduced when occurring as part of a larger expression. The locations where such reduction can occur are

defined by means of *evaluation contexts*. These are expressions containing a hole denoted by $-$ indicating a position where a reduction can take place. They are given by the grammar

$$\begin{aligned} E[-] ::= & - \mid EN \mid \text{succ } E \mid \text{pred } E \mid \text{cond } E \ N_1 \ N_2 \mid \\ & \text{seq } E \ N \mid \text{deref } E \mid \text{assign } E \ n \mid \text{assign } M \ E \mid \\ & \text{mkvar } M \ E \mid \text{mkvar } E \ M \mid \text{new } x \text{ in } E . \end{aligned}$$

The small-step semantics is then completed with the rule:

$$\frac{M \rightarrow N}{E[M] \rightarrow E[N]} .$$

Substitution

The substitution operation naturally extends to IA: it is done inductively on the structure of the term. For the block-variable case this gives:

$$\begin{aligned} (\text{new } x \text{ in } M) [N/y] &= \text{new } z \text{ in } M [z/x] [N/y] && \text{if } x \neq y, z \text{ fresh;} \\ (\text{new } x \text{ in } M) [N/x] &= \text{new } x \text{ in } M \end{aligned}$$

Capture-permitting substitution is defined similarly. The former equation becomes:

$$(\text{new } x \text{ in } M) \{N/y\} = \text{new } x \text{ in } M \{N/y\} \quad \text{if } x \neq y.$$

2.2 Higher-Order Grammars and the Safety Restriction

We present the safety restriction in the context of higher-order grammars as it was originally defined [KNU02]. We give a brief introduction to the concept of higher-order grammars. A more detailed introduction on the subject is de Miranda's thesis [dM06].

2.2.1 Higher-order grammars

We consider simple types over a single atom o . Given a set of typed symbols S , the set of **applicative terms** generated from S , written $\mathcal{A}(S)$ is defined as the closure of S under the application rule (*i.e.*, if $M : A \rightarrow B$ and $N : A$ are in $\mathcal{A}(S)$ then so is $MN : B$).

Definition 2.2.1. A *higher-order grammar* is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, where

- Σ is a ranked alphabet (in the sense that each symbol $f \in \Sigma$ has an arity $\text{arity}(f) \geq 0$) of *terminals*;
- \mathcal{N} is a finite set of typed *non-terminals*;
- S is a distinguished ground-type symbol of \mathcal{N} , called the start symbol;
- \mathcal{R} is a finite set of production (or rewrite) rules, one for each non-terminal $F : (A_1, \dots, A_n, o) \in \mathcal{N}$, of the form:

$$F z_1 \dots z_m \rightarrow e$$

where each z_i (called *parameter*) is a variable of type A_i and e is an applicative term of type o generated from the typed symbols in $\Sigma \cup \mathcal{N} \cup \{z_1 : A_1, \dots, z_m : A_m\}$.

We say that the grammar is *order- n* just in case the order of the highest-order non-terminal is n .

Higher-order grammars as generators of term tree languages

An applicative term generated from the terminals in Σ only (without non-terminals), and viewed as a Σ -labelled tree, is called a **value term**.

A higher-order grammar G defines a tree language denoted $L(G)$ defined as the set of *finite* value terms which can be seen as applicative terms over $\mathcal{N} \cup \Sigma$. It is formally defined as the set consisting of all the finite value terms that can be obtained by normalizing the start symbol S using the the reduction relation induced by the rewriting rules of G . This normalization can be done using different reduction strategies also called *derivation modes*. The three main ones are: outside-in (OI), inside-out (IO), and unrestricted. As the names suggest, in the OI derivation mode the outermost redex is reduced first; in IO mode the innermost redex is reduced first; and in unrestricted mode, no particular choice of redex is imposed. It can be shown that the OI derivation is sufficient in the sense that every value term obtained from an IO derivation can also be obtained from an OI derivation. The converse however does not hold [Dam82].

REMARK 2.2.1 In the literature [Dam82, dM06, KNU02], the ranked-alphabet Σ is usually restricted to terminals of order 1 at most so that each $f \in \Sigma$ of arity $r \geq 0$ is assumed to have type $(\underbrace{o, \dots, o}_r, o)$. The idea is that order-0 terminals correspond to leaves of the value tree and order-1 terminals correspond to nodes. Such restriction is however not necessary as one can just regard higher-order terminals as special nodes with additional constraints on the type of their children.

Higher-order grammars as word language generators

Higher-order grammars can be used as generators of word languages by imposing the following constraints on the set of terminals Σ :

- Σ contains a special symbol $e : o$,
- all other constant $f \in \Sigma$ are of type (o, o) .

The idea is that the type o represent the type of strings Σ^* , the symbol e marks the end of the word and a constant $f : (o, o)$ represents the operation that appends the letter ‘ f ’ as a prefix to a string.

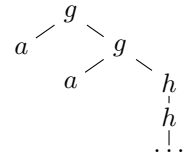
Higher-order grammars as tree generators

In order to generate infinite trees, higher-order grammars are specialized into a device called *recursion scheme*. A **higher-order recursion scheme** is a higher-order grammar where the set of rewrite rules is deterministic (*i.e.*, for each non-terminal $F \in \mathcal{N}$ there is exactly one production rule with F on the left-hand side).

A recursion scheme R defines a (potentially infinite) value tree denoted $\llbracket R \rrbracket$ obtained by unfolding its rewrite rules *ad infinitum*, replacing formal by actual parameters each time, starting from the start symbol S . Formally, $\llbracket R \rrbracket$ is defined as the least upper bound of the *schematological tree grammar* induced by R in the continuous algebra of ranked trees with the appropriate ordering [KNU02, dM06].

Example 2.2.1. Let G be the following order-2 recursion scheme:

$$\begin{aligned} S &\rightarrow H a \\ H z^o &\rightarrow F(g z) \\ F \phi^{(o,o)} &\rightarrow \phi(\phi(F h)) \end{aligned}$$



where the arities of the terminals g, h, a are 2, 1, 0 respectively. The tree generated by G is defined by the infinite term $g a (g a (h (h (h \dots))))$.

2.2.2 The safety restriction

Safety is a syntactic restriction for higher-order grammars introduced by Knapik et al. to study the Monadic Second Order (MSO) theory of infinite trees generated by higher-order pushdown automata [KNU02]. The safety restriction has appeared under different forms in the literature. The first formulation, due to Damm, appeared under the name *restriction of derived types* [Dam82]. De miranda's thesis contains a comparison of the two formulations [dM06]. The presentation given here follows that of Knapik et al. [KNU02].

Type homogeneity

We say that a type is **homogeneous** if it is o or if it is (A_1, \dots, A_n, o) with the condition that $\text{ord } A_1 \geq \text{ord } A_2 \geq \dots \geq \text{ord } A_n$ and each A_1, \dots, A_n is homogeneous [KNU02].

NOTATION 2.2.1 (Type partitioning) Suppose that $\overline{A_1}, \overline{A_2}, \dots, \overline{A_n}$ are n lists of types, where A_{ij} denotes the j th type in the list $\overline{A_i}$ and l_i the size of $\overline{A_i}$, then we write

$$A = (\overline{A_1} \mid \dots \mid \overline{A_n} \mid o)$$

to mean that

- A is the type $(A_{11}, A_{12}, \dots, A_{1l_1}, A_{21}, \dots, A_{2l_2}, \dots, A_{n1}, \dots, A_{nl_n}, o)$,
- $\forall i : \forall u, v \in A_i : \text{ord } u = \text{ord } v$,
- $\forall i, j. \forall u \in A_i. \forall v \in A_j. i < j \implies \text{ord } u > \text{ord } v$.

This implies in particular that A is homogenous. In other words, this notation partitions the A_{ij} s according to their order. Let $B = (\overline{B_1} \mid \dots \mid \overline{B_m} \mid o)$, we write $(\overline{A_1} \mid \dots \mid \overline{A_n} \mid B)$ as an abbreviation for $(\overline{A_1} \mid \dots \mid \overline{A_n} \mid \overline{B_1} \mid \dots \mid \overline{B_m} \mid o)$.

Definition

Definition 2.2.2 (Safe grammar). (All types are assumed to be homogeneous.) A term of order $k > 0$ is *unsafe* if it contains an occurrence of a parameter of order strictly less than k , otherwise the term is *safe*. An occurrence of an unsafe term t as a subexpression of a term t' is *safe* if it is in the context $\dots(ts)\dots$, otherwise the occurrence is *unsafe*. A grammar is **safe** if no unsafe term has an unsafe occurrence at a right-hand side of any production.

This definition is a bit opaque and does not seem to make a lot of sense at first. One can reformulate this definition in a slightly clearer way: A higher-order grammar G whose non-terminals are of homogeneous type is *unsafe* if and only if there is a rewrite rule $Fz_1 \dots z_m \rightarrow e$ where e contains a subterm that:

1. occurs in *operand* position in e ,
2. contains a parameter of order strictly less than its order.

(By “operand position” we mean “in the second position of some occurrence of the implicit application operator of the lambda calculus”.) A grammar is *safe* if it is not unsafe.

Example 2.2.2 ([KNU02]). Let $f : (o, o, o)$, $g, h : (o, o)$ and $a, b : o$ be Σ constants. The grammar of level 3 with non-terminals $S : o$ and $F : ((o, o), o, o, o)$ and production rules:

$$\begin{aligned} S &\rightarrow Fgab \\ F\varphi xy &\rightarrow f(F(F\varphi x)y(hy))(f(\varphi x)y) \end{aligned}$$

is not safe because the term $F\varphi x : (o, o)$ containing a variable of order 0 occurs at an operand position in the right-hand side expression of the second rule.

On the other hand, the grammar with the following production rules is safe:

$$\begin{aligned} S &\rightarrow G(ga)b \\ Gzy &\rightarrow f(G(Gzy)(hy))(fzy) \end{aligned}$$

Moreover it can be shown that these two recursion schemes are equivalent in the sense that they generate the same infinite tree.

Example 2.2.3. Take $H : ((o, o), o)$ and $f : (o, o, o)$; the following rewrite rules are unsafe (in each case we underline the unsafe subterm that occurs unsafely):

$$\begin{aligned} G^{(o,o)} x &\rightarrow H(f x) \\ F^{((o,o), o, o, o)} z x y &\rightarrow f(\overline{F}(\overline{F} z y) y(z x)) x \end{aligned}$$

Example 2.2.4. The order-2 grammar defined in Example 2.2.1 is unsafe.

2.2.3 Automata Characterization

Although very technical, the safety restriction for higher-order recursion schemes as appealing machine characterization. Knapik, Niwiński and Urzyczyn showed that for generating infinite ranked trees, safe higher-order recursion scheme are as expressive as *higher-order pushdown automata* (PDA) [KNU02].

A **pushdown automaton** (PDA) is an infinite-state system transition system that can manipulate the content of a stack when performing a transition. Higher-order pushdown automata were introduced as a generalization of PDA [Mas76]. Instead of manipulating a simple stack, a higher-order PDA manipulates iterated stacks. An order-1 PDA is an ordinary PDA, an order-2 PDA manipulates order-2 stacks which are stacks of order-1 stacks. In addition to the usual push and pop transitions of a PDA, an order-2 PDA has order-2 variants: a $push_2$ operation that duplicates the top order-1 stack, and a pop_2 that pops the entire top order-1 stack. This definition generalizes to any order $n \in \mathbb{N}$.

Theorem 2.2.1 (Knapik, Niwiński and Urzyczyn, [KNU02]). *Let L be a Σ -labelled term tree language. L is the language of a safe order- n grammar (using the OI derivation) if and only if it is accepted by an order- n pushdown automaton.*

So in particular, a (potentially) infinite tree t is generated by a safe order- n recursion scheme if and only if it is accepted by an order- n pushdown automaton.

A similar characterization has subsequently been obtained for unrestricted grammars: Hague, Murawski, Ong and Serre have introduced a new kind of pushdown automata called *collapsible pushdown automata* (CPDA) and showed their equivalence with unrestricted higher-order grammars. The internal structure manipulated by a CPDA is a stack in which every symbol has a link pointing to some other substacks situated below it. There is an additional stack-operation called *collapse* whose effect is to replace the content of the stack by the sub-stack indicated by the link attached to the top symbol of the stack.

Theorem 2.2.2 (Hague, Murawski, Ong and Serre, [HMOS08]). *Let t be an potentially infinite tree. t is generated by an order- n recursion scheme if and only if it is accepted by an order- n collapsible pushdown automaton.*

We have defined higher-order grammars as generators of word languages and trees. Thanks to the machine characterization, it is possible to define the notion of graph generated by a higher-order grammars: the graph generated by a grammar is defined as the configuration graph of the corresponding collapsible higher-order pushdown automaton. In particular, graph generated by a safe grammar is the configuration graph of the corresponding higher-order PDA.

2.2.4 Expressivity

Higher-order PDA/grammars can be used as generating device for word-languages, trees, or graphs, thus inducing strict infinite hierarchies as the order of the PDA varies. For word-languages this is known as the Maslov hierarchy: orders 0, 1 and 2 correspond respectively to the regular, context-free and indexed languages. For trees, orders 0, 1 and 2 are respectively the regular, algebraic and hyperalgebraic trees.

2.2.5 Is safety a genuine restriction?

The consequences that the safety constraint has on the expressivity of higher-order grammars are not completely understood. A partial answer has been given for word languages: Aehlig, De Miranda and Ong showed that up to order 2, there is no intrinsically unsafe word language [AdMO05b]: any word language generated by an unsafe order-2 grammar can also be generated by some (potentially non-deterministic) order-2 safe grammar. For trees, Urzyczyn conjectured [dM06] that safety constrains expressivity. He even proposed a tree called Urzyczyn's tree that is generated by an unsafe order-2 recursion scheme that he conjectured to not be generated by any safe order-2 recursion scheme. At present, no one has managed to prove or disprove this conjecture.

A similar question can be asked from a verification point of view: are the structures generated by safe higher-order grammars easier to verify than those generated by unrestricted grammars? The reason why the safety constraint was introduced in the first place was precisely to be able to show that the generated trees have decidable Monadic Second Order (MSO) theories [KNU02]. It turns out in fact that this result holds in the general unrestricted case as Ong later showed [Ong06a]:

Theorem 2.2.3. *The modal mu-calculus model checking problem for trees generated by order- n recursion schemes is n -EXPTIME complete for each $n \geq 0$.*

This result implies that these trees have decidable MSO theories since the two logics are equi-expressive over trees. The proof of this theorem is based on a game semantic argument based on the theory of traversals (that will be presented in Chapter 4) which radically differs from the argument used by Knapik et al. for the case of safe grammars [KNU02]. A generalization of Theorem 2.2.3 for graphs was later obtained by Hague et al. [HMOS08]:

Theorem 2.2.4. *For each $n \geq 0$, the modal mu-calculus model checking problem for configuration graphs of order- n collapsible pushdown systems is n -EXPTIME complete.*

For graphs, the MSO logic is strictly more expressive than the modal mu-calculus. In the same paper it is shown that MSO theories of collapsible pushdown graphs are undecidable while those of pushdown graphs are decidable [HMOS08]. Hence from a verification point of view, safety can indeed be considered as a genuine constraint.

2.2.6 Higher-order grammars and the simply typed lambda calculus

Higher-order grammars are essentially closed simply type lambda terms of ground type extended with recursion and generated from function symbols Σ . We now make this correspondence explicit. The syntax of the simply typed lambda calculus is extended with the mutual recursion operator Y_{mut} :

$$(Y_{\text{mut}}) \frac{\Gamma \vdash M_1 : A \rightarrow A_1 \quad \Gamma \vdash M_q : A \rightarrow A_q}{\Gamma \vdash Y_{\text{mut}}(M_1, \dots, M_q) : A_1} \quad A = A_1 \times \dots \times A_q, q \geq 0$$

whose semantics is given by the rule $Y_{\text{mut}}(M_1 \dots M_q) \rightarrow \pi_1(Y\langle M_1 \dots M_q \rangle)$ where π_1 denotes the projection of a q -tuple to its first component and Y denotes the extension of the usual Y -combinator to product types with a semantics given by the rule:

$$Y\langle M_1, \dots, M_q \rangle \rightarrow \langle M_1(Y\langle M_1, \dots, M_q \rangle), \dots, M_q(Y\langle M_1, \dots, M_q \rangle) \rangle .$$

We write $\Lambda_{\rightarrow}^{mut}(\Sigma)$ to denote the simply typed calculus extended with mutual recursion and Σ -constants.

A higher-order grammar can then be equivalently expressed as a closed simply typed term from $\Lambda_{\rightarrow}^{mut}(\Sigma)$. The first direction is trivial: Take a grammar $\langle \Sigma, \mathcal{N}, \mathcal{R}, F_0 \rangle$ where $\mathcal{R} = \{F_0, \dots, F_q\}$ for some $q \geq 0$. We convert each rule $F_i x_1 \dots x_n \rightarrow e$ into a lambda term $\widetilde{F}_i \equiv \lambda F_0 \dots F_q x_1 \dots x_n. e$. The grammar can then be equivalently formulated as the term $Y_{mut}(\widetilde{F}_0, \dots, \widetilde{F}_q)$.

Conversely, any lambda term using mutual recursion and Σ -constants can be equivalently formulated as a higher-order grammar. The conversion proceeds inductively over the syntax of the η -long normal form of the term. The algorithm **LmdToHOGr** is described in Table 2.5 in an ML-like syntax. The function **LmdToHOGr** take a closed term from $\Lambda_{\rightarrow}^{mut}(\Sigma)$ and returns an equivalent higher-order grammar. The variables \mathcal{N} and \mathcal{R} contain respectively the list of non-terminals and rules created so far. The auxiliary function **createRules** creates the rules for an open lambda-term, adds them to the set \mathcal{R} and returns an applicative term in $\mathcal{A}(\mathcal{N} \cup \Sigma)$ that is equivalent to the given lambda-term. Observe that since we work on the η -long normal form, the right-hand side of each generated rule is indeed of ground type, as required by the definition of higher-order grammars. We use the symbol $@$ to represent the application of two applicative terms.

Input: a closed lambda-term $\vdash_{\Sigma} M : T$ in η -normal form potentially using mutual recursion.

Output: an equivalent higher-order grammar scheme $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$.

```

let LmdToHOGr( $\vdash_{\Sigma} M : T$ )
  let createRules :  $\Lambda_{\rightarrow}^{mut}(\Sigma) \rightarrow \mathcal{A}(\mathcal{N} \cup \Sigma) = \text{fun}$ 
    |  $\Gamma \vdash \alpha : T$  with  $\alpha \in \Gamma \cup \Sigma$        $\rightarrow$   $\alpha$ 
    |  $\Gamma \vdash M_0 \dots M_q : o$                    $\rightarrow$  createRules( $\Gamma \vdash M_0 : T_0$ )
                                              @ ...
                                              @createRules( $\Gamma \vdash M_q : T_q$ )
    |  $\overline{x} : \overline{A} \vdash \lambda \overline{y}^{\overline{B}}. M : (\overline{B}, o)$        $\rightarrow$  let  $t = \text{createRules}(\overline{x} : \overline{A}, \overline{y} : \overline{B} \vdash M : o)$ 
                                              and  $F$  be a fresh non-terminal name in
                                               $\mathcal{R} \leftarrow "F \overline{x} \overline{y} \rightarrow t" :: \mathcal{R}$ 
                                               $\mathcal{N} \leftarrow "F : (\overline{A}, \overline{B}, o)" :: \mathcal{N}$ 
                                              " $F \overline{x}$ "
    |  $\overline{x} : \overline{A} \vdash Y_{mut}(M_1, \dots, M_q) : B_1$   $\rightarrow$  // where  $M_i : B_i$  for  $i \in \{1..q\}$ 
                                              for  $i = 1 \dots q$  do
                                              createRules( $\overline{x} : \overline{A} \vdash M_i : B_i$ )
                                              let " $F_i \overline{x} f_1 \dots f_q \overline{y}_i \rightarrow t_i$ "  $\leftarrow$  hd  $\mathcal{R}$  in
                                               $\mathcal{R} \leftarrow "\widehat{F}_i \overline{x} \overline{y}_i \rightarrow t_i[\widehat{F}_1 \overline{x} / f_1] \dots [\widehat{F}_q \overline{x} / f_q]"$ 
                                               $::$  tail  $\mathcal{R}$ 
                                               $\mathcal{N} \leftarrow "\widehat{F}_i : (\overline{A}, B_i)" ::$  tail  $\mathcal{N}$ 
                                              done
                                              " $\widehat{F}_1 \overline{x}$ "
  in
   $\mathcal{N}, \mathcal{R} \leftarrow [], []$ 
   $appterm \leftarrow \text{createRules}(\vdash M : T)$ 
   $\langle \Sigma, "S \rightarrow appterm" :: \mathcal{N}, "S : o" :: \mathcal{R}, S \rangle$ 

```

Table 2.5: Algorithm **LmdToHOGr** converting a mutually recursive lambda term into a higher-order grammar.

2.3 Game Semantics

Game semantics is a very powerful paradigm for giving models of programming languages. It was the first kind of semantics able to provide a *fully abstract model* of the language PCF, a result which was subsequently extended to other languages. In a nutshell, the term “full abstraction” means that the model provides a faithful mathematical characterizations of the language. A natural way to give a semantic account of a language consists therefore in giving a game semantic characterization of it. A question that we will try to answer in this thesis is how does a syntactic restriction such as *safety* impact on the on the game model of a language? A substantial part of this thesis is devoted to this question (Chapter 4 and 6).

This chapter introduces the basic notions of game semantics including the categorical interpretation, the game interpretation of PCF and IA, and the full abstraction results. It concludes by giving a brief summary of some important results in *Algorithmic game semantics*. For an introduction, we recommend the tutorial by Samson Abramsky [AM98b] on which this chapter is based. Many details and proofs will be omitted; we refer the reader to other literature [HO00, AMJ94] for a complete description of game semantics. The reader familiar with game semantics may very well consider skipping this chapter altogether as all the definitions and notations introduced here are standard.

2.3.1 History

Game semantics finds its origin in various work. Paul Lorenzen introduced a game semantics for logic in the 1950s to study intuitionistic logic [Lor61]. In this setting, the notion of logical truth is modeled using game theoretic concepts such as the existence of a winning strategy. Four decades later, Andreas R. Blass established a connection with linear logic. At the same time, Samson Abramsky and Radha Jagadeesan [AJ92] on one hand, Martin Hyland and Luke Ong [HO93] on the other hand, used game semantics to prove full completeness of Multiplicative Linear Logic (MLL). Game semantics has then emerged as a new paradigm for the study of formal models for programming languages. The idea is to model the execution of a program as a game played by two protagonists. Two players are involved in these game: the Opponent, who represents the environment, and the Proponent, who representing the program. The meaning of the program is then modeled by a strategy for the Proponent.

Subsequently, these game-based models have been used to give a solution to the long-standing problem of “Full abstraction of PCF” [AMJ94, HO00, Nic94]. This result has then led to applications in the field of software verification [GM00] opening-up a new field called Algorithmic Game Semantics [Abr01a].

Model of programming languages

Before the 1980s, there were many approaches to define models for programming languages. Among the successful ones, there were the axiomatic, operational and denotational semantics.

- Operational semantics gives a meaning to a program by describing the behaviour of a machine executing the program. It is defined formally by giving a state transition system.
- Axiomatic semantics defines the behaviour of the program through the use of axioms and is used to prove program correctness by static analysis of the code of the program.
- The denotational semantics approach consists in mapping a program to a mathematical structure having good properties such as compositionality. This mapping is achieved by structural induction on the syntax of the program.

In the 1990s, three different independent research groups: Samson Abramsky, Radhakrishnan Jagadeesan and Pasquale Malacaria [AMJ94], Martin Hyland and Luke Ong [HO00] and Nickau [Nic94] introduced game semantics, a new kind of semantics, in order to solve a long standing problem in the semanticists community, namely finding a fully abstract model for PCF.

The problem of full abstraction for PCF

The problem of the Full Abstraction for PCF goes back to the 1970s. Scott constructed a model of PCF based on domain theory [Sco93] which gives a sound interpretation of observational equivalence: if two terms have the same domain theoretic interpretation then they are observationally equivalent. However the converse is not true: there exist two PCF terms which are observationally equivalent but have different domain theoretic denotations. We say that the model is not *fully abstract*.

The key reason why the domain theoretic model is not fully abstract is that the parallel-or operator defined by the following truth table

p-or	\perp	tt	ff
\perp	\perp	tt	\perp
tt	tt	tt	tt
ff	\perp	tt	ff

is not definable by any PCF term. Indeed, it is possible to define two different PCF terms that have the same behaviour except when applied to a term computing p-or. Since p-or is not definable in PCF, these two terms will have the same denotation. Hence the model is not fully abstract.

It is possible to “patch” PCF by adding the operator p-or, the resulting language “PCF+p-or” is fully-abstracted by Scott domain theoretic model [Plo77]. The language we are now dealing with, however, is strictly more powerful than PCF since it allows parallel execution of commands whereas PCF only permits sequential execution.

Another approach involves the elimination of the undefinable elements (like p-or) by strengthening the conditions on the function used in the model. This approach was followed by Berry who gave a model based on stable functions [Ber78, Ber79]. Stable functions are a class of functions smaller than the class of strict and continuous function. Unfortunately this approach did not succeed.

Full abstract models for PCF were found at the same time and independently by three research teams: Abramsky, Jagadeesan and Malacaria [AMJ94], Hyland and Ong [HO00] and Nickau [Nic94]. These three approaches are all based on game semantics.

The game semantics approach has then been adapted to other varieties of programming paradigms including languages with stores (Idealized Algol), call-by-value [HY99, AM98a] and call-by-name, general references [AHM98], polymorphism [AJ05], control features (continuation and exception), non determinism, concurrency. In all these cases, the game semantics model led to a syntax-independent fully abstract model of the corresponding language.

2.3.2 Definitions

We now introduce formally the notion of game that will be used in the following sections to give a model of the programming languages PCF and Idealized Algol. The games we are interested in are two-players games. The players are named O for *Opponent* and P for *Proponent*. The game played by O and P is constrained by the *arena*. The arena defines the possible moves of the game. By analogy with real board games, the arena represents the board together with the rules that tell players how they can make their moves on the board. The analogy with board game will not go beyond that. Instead, it is better to regard our games as dialogs between two players: player O interviews player P. P’s goal is to answer the initial question asked by O. P can also ask questions to O if he needs more precision about O’s initial question. Again, O can ask further question to P. This induces a flow of questions and answers between O and P that can continue possibly forever. In game semantics the attention is given to the study of this flow of questions and answers, the notion of winner of the game is not a concern.

2.3.2.1 Arenas

Our games have two kinds of moves: the questions and the answers. We also distinguish moves played by O and those played by P. An arena is represented by a directed acyclic graph (DAG)

whose nodes correspond to question moves and leaves correspond to answer moves. It is formally defined as follows.

Definition 2.3.1 (Arena). An arena is a structure $\langle M, \lambda, \vdash \rangle$ where:

- M is the set of possible moves;
- (M, \vdash) is a directed acyclic graph;
- $\lambda : M \rightarrow \{O, P\} \times \{Q, A\}$ is a labelling function indicating whether a given move is a question or an answer and whether it can be played by O or P.
 $\lambda = [\lambda^{OP}, \lambda^{QA}]$ where $\lambda^{OP} : M \rightarrow \{O, P\}$ and $\lambda^{QA} : M \rightarrow \{Q, A\}$.
 - If $\lambda^{OP}(m) = O$, we call m an O-move otherwise m is a P-move. $\lambda^{QA}(m) = Q$ indicates that m is a question otherwise m is an answer.
 - a leaf l of the DAG (M, \vdash) satisfies $\lambda^{QA}(l) = A$ and a node n satisfies $\lambda^{QA}(n) = Q$.
- The DAG (M, \vdash) respects the following condition:
 - (e1) The roots are O-moves: for any root r of (M, \vdash) , $\lambda^{OP}(r) = O$;
 - (e2) enablers are questions: $m \vdash n \implies \lambda^{QA}(m) = Q$;
 - (e3) a player's move must be justified by a move played by the other player: $m \vdash n \implies \lambda^{OP}(m) \neq \lambda^{OP}(n)$.

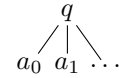
For will abbreviate the set $\{O, P\} \times \{Q, A\}$ as $\{OQ, OA, PQ, PA\}$. $\bar{\lambda}$ denotes the labelling function obtained from λ after swapping players:

$$\begin{aligned} \overline{\lambda(m)} &= OQ \iff \lambda(m) = PQ \\ \text{and } \overline{\lambda(m)} &= OA \iff \lambda(m) = PA \end{aligned}$$

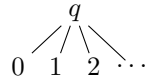
The roots of the DAG (M, \vdash) are called the *initial moves*. Other moves must be enabled by some other question move. The edges of the DAG induces the enabling relation between moves.

The simplest possible arena, written **1**, is the arena with an empty set of moves.

Example 2.3.1 (The flat arena). Let A be any countable set, then the flat arena over A is defined to be the arena $\langle M, \lambda, \vdash \rangle$ such that M has one move q with $\lambda(q) = OQ$ and for each element in A , there is a corresponding move a_i in M with $\lambda(a_i) = PA$ for some $i \in \mathbb{N}$. The enabling relation \vdash is defined to be $\{q \vdash a_i \mid i \in \mathbb{N}\}$. This arena is represented by the tree



represent the moves and edges represent the enabling relation. In the rest of this thesis we will just write \mathbb{N} to mean the flat arena over \mathbb{N} :



Once the arena has been defined, the bases of the game are set and the players have something to play with. We now need to describe the state of the game, for that purpose we introduce *justified sequences of moves*:

Definition 2.3.2 (Justified sequence of moves). A justified sequence is a sequence of moves s together with an associated sequence of pointers. Any move m in the sequence that is not initial has a pointer that points to a previous move n that enables it (*i.e.*, $n \vdash m$).

(A more formal definition is to regard a justified sequence as a sequence of pairs, each pair encoding an element of the sequence together with an index indicating the position where the element points to.)

Since initial moves are all O-moves, the first move of a justified sequence is necessarily an O-move.

CONVENTION 2.3.1 Justification pointers are graphically represented by arrows as in the following example:

$$q^4 \overset{\curvearrowright}{\leftarrow} q^3 \overset{\curvearrowright}{\leftarrow} q^2 \overset{\curvearrowright}{\leftarrow} q^3 \overset{\curvearrowright}{\leftarrow} q^2 \overset{\curvearrowright}{\leftarrow} q^1 .$$

We will sometimes omit the justification pointers altogether if they do not play any role in the argument.

NOTATION 2.3.1 We write st or sometimes $s \cdot t$ to denote the sequences obtained by concatenating s and t . The empty sequence is written ϵ . Given a justified sequence $s = m_1 \cdot m_2 \dots m_n$ (where pointers are not represented) we write $s_{\leq m_i}$ for $m_1 \cdot m_2 \dots m_i$, the prefix sequence of s up to the move m_i ; and $s_{< m_i}$ for $m_1 \cdot m_2 \dots m_{i-1}$.

Justified sequences of moves will be used to record the history of all the moves that have been played so far in the (yet to be defined) game. Two particular subsequences called the *P-view* and the *O-view* are of interest. These subsequences correspond to local restricted views that each player has of the history of the game.

Definition 2.3.3 (View). Given a justified sequence of moves s , the **Proponent view** (P-view) written $\lceil s \rceil$ is defined by induction as follows:

$$\begin{aligned} \lceil \epsilon \rceil &= \epsilon, \\ \lceil s \cdot m \rceil &= \lceil s \rceil \cdot m && \text{if } m \text{ is a P-move,} \\ \lceil s \cdot m \rceil &= m && \text{if } m \text{ is initial (O-move),} \\ \lceil s \cdot m \cdot t \cdot n \rceil &= \lceil s \rceil \cdot m \cdot n && \text{if } n \text{ is a non initial O-move.} \end{aligned}$$

The **O-view** $\lfloor s \rfloor$ is defined similarly:

$$\begin{aligned} \lfloor \epsilon \rfloor &= \epsilon, \\ \lfloor s \cdot m \rfloor &= \lfloor s \rfloor \cdot m && \text{if } m \text{ is a O-move,} \\ \lfloor s \cdot m \cdot t \cdot n \rfloor &= \lceil s \rceil \cdot m \cdot n && \text{if } n \text{ is a P-move.} \end{aligned}$$

2.3.2.2 Games

Not all justified sequences are of interest for the notion of games that we use. We call *legal position* justified sequences that satisfy two additional conditions: alternation and visibility. Alternation says that players O and P play alternatively. Visibility expresses that each non-initial move is justified by a move situated in the local context at that point. Formally:

Definition 2.3.4 (Legal position). A legal position is a justified sequence of move s respecting the following constraints:

- *Alternation*: For any subsequence $m \cdot n$ of s , $\lambda^{OP}(m) \neq \lambda^{OP}(n)$.
- *Visibility*: For any subsequence tm of s where m is not initial, if m is a P-move then m points to a move in $\lceil s \rceil$ and if m is a O-move then m points to a move in $\lfloor s \rfloor$.

The set of legal position of an arena A is denoted by L_A .

We say that a move n is hereditarily justified by a move m if there is a sequence of move m_1, \dots, m_q such that:

$$m \vdash m_1 \vdash m_2 \vdash \dots \vdash m_q \vdash n$$

If a move has no justification pointer, we say that it is an *initial move* (in that case it must be a root of the DAG of the arena).

Suppose that n is an occurrence of a move in the sequence s then $s \upharpoonright n$ denotes the subsequence of s containing all the moves hereditarily justified by n . Similarly, $s \upharpoonright I$ denotes the subsequence of s containing all the moves hereditarily justified by moves in I .

Definition 2.3.5 (Game). A game is a structure $\langle M, \lambda, \vdash, P \rangle$ such that

- $\langle M, \lambda, \vdash \rangle$ is an arena;
- P is called the set of valid positions, it is:
 - a non-empty prefix closed subset of the set of legal positions,
 - closed by initial hereditary projection: if s is a valid position then for any set I of occurrences of initial moves in s , $s \upharpoonright I$ is also a valid position.

The empty arena **1** together with the empty set of valid positions defines the simplest possible game, written **1**.

Example 2.3.2. Consider the flat arena \mathbb{N} . The set of valid position $P = \{\epsilon, q\} \cup \{q \cdot a_i \mid i \in \mathbb{N}\}$ defines a game on the arena \mathbb{N} .

2.3.2.3 Constructions on games

We now define game constructors that will be useful later on.

Consider the two functions $f : A \rightarrow C$ and $g : B \rightarrow C$, we write $[f, g]$ to denote the pairing of f and g defined on the direct sum $A + B$. Given a game A with a set of moves M_A , we use the projection operator $s \upharpoonright A$ to denote the subsequence of s consisting of all moves in M_A . Although this notation conflicts with the hereditary projection operator, it should not cause any confusion.

Tensor product Given two games A and B the tensor product $A \otimes B$ is defined as:

$$\begin{aligned} M_{A \otimes B} &= M_A + M_B \\ \lambda_{A \otimes B} &= [\lambda_A, \lambda_B] \\ \vdash_{A \otimes B} &= \vdash_A \cup \vdash_B \\ P_{A \otimes B} &= \{s \in L_{A \otimes B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}. \end{aligned}$$

In particular, n is initial in $A \otimes B$ if and only if n is initial in A or B . And $m \vdash_{A \otimes B} n$ holds if and only if $m \vdash_A n$ or $m \vdash_B n$ holds.

Function space The game $A \multimap B$ is defined as follows:

$$\begin{aligned} M_{A \multimap B} &= M_A + M_B \\ \lambda_{A \multimap B} &= [\overline{\lambda_A}, \lambda_B] \\ \vdash_{A \multimap B} &= \vdash_A \cup \vdash_B \cup \{(m, n) \mid m \text{ initial in } B \wedge n \text{ initial in } A\} \\ P_{A \multimap B} &= \{s \in L_{A \otimes B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B \in P_B\}. \end{aligned}$$

Cartesian product The game $A \& B$ is defined as follows:

$$\begin{aligned} M_{A \& B} &= M_A + M_B \\ \lambda_{A \& B} &= [\lambda_A, \lambda_B] \\ \vdash_{A \& B} &= \vdash_A \cup \vdash_B \\ P_{A \& B} &= \{s \in L_{A \otimes B} \mid s \upharpoonright A \in P_A \wedge s \upharpoonright B = \epsilon\} \\ &\quad \cup \{s \in L_{A \otimes B} \mid s \upharpoonright A \in P_B \wedge s \upharpoonright A = \epsilon\}. \end{aligned}$$

Note that a play of the game $A \& B$ is either a play of A or a play of B , whereas a play of the game $A \otimes B$ may be an interleaving of plays on A and plays on B .

2.3.2.4 Representation of plays

Plays of the game are usually represented in a table diagram. The columns of the table correspond to the different components of the arena and each row corresponds to one move in the play. The first row always represents an O-move, this is because O is the only player who can open a game (since roots of the arena are O-moves).

For example the play $q \overleftarrow{q} 8 9$ on the game $\mathbb{N} \multimap \mathbb{N}$ is represented by the following diagram:

\mathbb{N}	\multimap	\mathbb{N}
	q	O
q		P
8		O
	9	P

Whenever necessary the diagram also shows the justification pointers.

2.3.2.5 Strategy

When a player has to play during the game, he may face several choices for his next move. A strategy is a guide telling the player which move to make when the game is in a given position. Formally, it is a partial function mapping legal positions where P has to play to P-moves.

Definition 2.3.6. A **strategy** for player P on a given game $\langle M, \lambda, \vdash, P \rangle$ is a non-empty set of even-length positions from P such that:

1. if $sab \in \sigma$ then $s \in \sigma$ (*no unreachable position*);
2. if $sab, sac \in \sigma$ then $b = c$ and b has the same justifier as c (*determinacy*).

The idea is that the presence of the even-length sequence sab in σ tells the player P that whenever the game is in position s and player O plays the move a then it must respond by playing the move b . The first condition ensures that the strategy σ only considers positions that the strategy itself could have led to in a previous move. The second condition in the definition requires that this choice of move is deterministic (*i.e.*, there is a function f from the set of odd length position to the set of moves M such that $f(sa) = b$).

For any game A , the smallest possible strategy is called the *empty strategy* and written \perp . It is formally defined by $\{\epsilon\}$, which corresponds to a strategy that never responds.

REMARK 2.3.1 There is an alternative definition for strategies in which a prefix-closed set is used as opposed to the above definition which relies on *even-length prefix*-closed sets. If σ denotes a strategy in the sense of Definition 2.3.6 then the corresponding strategy in the alternative definition is given by $\sigma \cup \text{dom}(\sigma)$ where $\text{dom}(\sigma)$ is the domain of σ defined as

$$\text{dom}(\sigma) = \{sa \in P_A^{\text{odd}} \mid \exists b. sab \in \sigma\}.$$

Informally, think of $\text{dom}(\sigma)$ as the collection of all paths in the sub-tree induced by the strategy in the game tree of all possible moves.

Copy-cat strategy For any arena A there is a strategy on the game $A \multimap A$ called the *copy-cat strategy*. We write A_1 and A_2 to denote the first and second copies of the arena A in the game $A \multimap A$. If A is the arena A_1 then A^\perp denotes the arena A_2 and conversely.

Let A be one of the arena A_1 or A_2 . The copy-cat strategy operates as follows: whenever P has to respond to an O-move played in A , it first replicates this move into the arena A^\perp . O then responds in A^\perp and finally P replicates O's response back to A .

It is formally defined by:

$$\text{id}_A = \{s \in P_{A \multimap A}^{\text{even}} \mid \forall t \leq^{\text{even}} s. t \upharpoonright A_1 = t \upharpoonright A_2\}$$

where P_A^{even} denotes the set of valid positions of even length in the game A and $t \leq^{\text{even}} s$ denotes that t is an even length prefix of s .

The copy-cat strategy is also called *identity strategy* since it is the identity for strategy composition as we will see in the next paragraph.

Example 2.3.3. The copy-cat strategy on \mathbb{N} is given by the following generic play:

$$\begin{array}{ccc} \mathbb{N} & \multimap & \mathbb{N} \\ & & q \\ q & & \\ n & & \\ & & n \end{array}$$

Note that we introduced this type of diagram in the first place in order to represent plays but, as we can see here, whenever the represented play is general enough, the diagram can be used to represent strategies.

The copy-cat strategy on $\mathbb{N} \multimap \mathbb{N}$ is given by the following diagram:

$$\begin{array}{ccccc} (\mathbb{N} \multimap \mathbb{N}) & \multimap & (\mathbb{N} \multimap \mathbb{N}) & & \\ & & q & & \\ & q & & & \\ q & & & q & \\ & & m & & \\ m & & & & \\ & n & & & \\ & & n & & \end{array}$$

2.3.2.6 Composition

Game semantics is used to give a model of higher-order functional languages, and in particular of the simply typed lambda calculus. It is well-known that any such model underlies a cartesian closed category [Cro93]. The notion of morphism composition in the category is incarnated by a notion of strategy composition. This feature is essential as we will see that computing the denotation of a composite program boils down to composing the denotation of its constituent programs.

Definition 2.3.7 (Interaction sequence). Let u be a sequence of moves from games A , B and C together with justification pointers from all moves except those initial in C . The **projection** of s on the game $A \multimap B$, written $u \upharpoonright A, B$ is the subsequence of s obtained by removing from u the moves in C and pointers to moves in C . The projection on $B \multimap C$ is defined similarly.

An **interaction sequence** is a sequence of moves with pointers from A , B and C such that $u \upharpoonright A, B$ and $u \upharpoonright B, C$ are legal positions of the game $A \multimap B$ and $B \multimap C$ respectively. We write $\text{Int}(A, B, C)$ for the set of all such sequences.

We define the projection on the game $A \multimap C$ as follows: $u \upharpoonright A, C$ is the subsequence of u consisting of the moves from A and C with some additional pointers. We add a pointer from $a \in A$ to $c \in C$ whenever a points to some move $b \in B$ itself pointing to c . All the pointers to moves in B are removed.

Given two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, the **interaction** $\sigma \parallel \tau$ of σ and τ is defined as

$$\sigma \parallel \tau = \{u \in \text{Int}(A, B, C) \mid u \upharpoonright A, B \in \sigma \wedge u \upharpoonright B, C \in \tau\} .$$

Definition 2.3.8 (Strategy composition). Let $\sigma : A \multimap B$ and $\tau : B \multimap C$ be two strategies. The **composite** $\sigma; \tau$ is defined as:

$$\sigma; \tau = \{u \upharpoonright A, C \mid u \in \sigma \parallel \tau\} .$$

This way of composing strategies is often summarized as performing “parallel composition plus hiding” as defined in the trace semantics of CSP [Hoa83].

It can be verified that composition is well-defined, associative and that the copy-cat strategy id_A is the identity for composition [HO00].

2.3.2.7 Constraint on strategies

Different classes of strategies will be considered depending on the features of the language that we want to model. Here is a list of common restrictions that we will consider:

- *Well-bracketing*: We call *pending question* the last question in a sequence that has not been answered. A strategy σ is well-bracketed if for every play $s \cdot m \in \sigma$ where m is an answer, m points to the pending question in s .
- *History-free strategies*: a strategy is history-free if the Proponent’s move at any position of the game where he has to play is determined by the last move of the Opponent. In other words, the history prior to the last move is ignored by the Proponent when deciding how to respond.
- *History-sensitive strategies*: The Proponent follows a history-sensitive strategy if he needs to have access to the full history of the moves in order to decide which move to make.
- *Innocence*: In these strategies, the Proponent determines his next move based solely on a restricted view of the history of the play, namely the P-view at that point. It always plays the same move when the game is in a given P-view. Innocence plays an important role in the modeling of higher-order functional languages.

The formal definition of innocence is:

Definition 2.3.9 (Innocence). Given positions $sab, ta \in L_A$ where sab has even length and $\lceil sa \rceil = \lceil ta \rceil$, there is a unique extension of ta by the move b together with a justification pointer such that $\lceil sab \rceil = \lceil tab \rceil$. We write this extension $\text{match}(sab, ta)$.

The strategy $\sigma : A$ is *innocent* if and only if:

$$\left(\begin{array}{l} \lceil sa \rceil = \lceil ta \rceil \\ sab \in \sigma \\ t \in \sigma \wedge ta \in P_A \end{array} \right) \implies \text{match}(sab, ta) \in \sigma .$$

Since the next move is determined by the P-view, an innocent strategy induces a partial function mapping P-views to P-moves called the *view function*. Not every partial function from P-views to P-moves gives rise to an innocent strategy, however. (Hyland and Ong [HO00] gave a sufficient condition.)

2.3.3 On the necessity of justification pointers

For any legal justified sequence of moves s , we write $?(s)$ for the subsequence of s obtained by keeping only the unanswered questions in s . It is easy to check that if s satisfies alternation then so does $?(s)$.

Lemma 2.3.1. *If $s \cdot q$ is a legal position (i.e., a justified sequence satisfying visibility and alternation) satisfying well-bracketing where q is a non-initial question then q points in $?(s)$.*

Proof. By induction on the length of $s \cdot q$. The base case $s = \epsilon$ is trivial. Let $s = s' \cdot q$, where q is not initial.

Suppose q is a P-move. We prove that q cannot point to an O-question that has been answered. Suppose that an O-move q' occurs before q and is answered by the move a also occurring before q . Then we have $s = s_1 \cdot q'^O \cdot s_2 \cdot a^P \cdot s_3 \cdot q^P$ where a is justified by q' . a is not in the P-view $\lceil s_{<q} \rceil$. Indeed this would imply that some O-move occurring in s_3 points to a , but this is impossible since answer

moves are not enablers. Hence the move a must be situated underneath an O-to-P link. Let m denote the link's origin, the P-view of s has the following form: $\ulcorner s \urcorner = \ulcorner s_1 \cdot q'^O \cdot s_2 \cdot a^P \dots m^O \urcorner \dots q^P$ where m is an O-move pointing before a .

If m is an answer move then it must point to the last unanswered move (the last move in $?(s_{<m})$). If m is a question move then it is not initial since there is a link going from m . Therefore by the induction hypothesis, m must point to a move in $?(s_{<m})$.

Since s is well bracketed, all the questions in the segment $q' \dots a$ are answered. Therefore since m points to an unanswered question occurring before a , m must point to a move occurring strictly before q' . Consequently q' does not occur in the P-view $\ulcorner s \urcorner$. By visibility, q must point in the P-view $\ulcorner s \urcorner$ therefore q does not point to q' .

A similar argument holds if q is an O-move. \square

This means that in a well-bracketed legal position $s \cdot m$ where m is not initial, m 's justifier is a question occurring in $?(s)$. Also if m is an answer then its justifier is precisely the *last* question in $?(s)$. Furthermore, if m is a P-move then by visibility it should point to an unanswered question in $\ulcorner m \urcorner$ therefore it should also point in $?(\ulcorner m \urcorner)$. Similarly, if m is a non initial O-move then it points in $?(\ulcorner m \urcorner)$.

Lemma 2.3.2. *Let s be a legal well-bracketed position.*

- (i) *If $s = \epsilon$ or if the last move in s is not a P-answer then $?(\ulcorner s \urcorner) = \ulcorner ?(s) \urcorner$;*
- (ii) *If $s = \epsilon$ or if the last move in s is not an O-answer then $?(\ulcorner s \urcorner) = \ulcorner ?(s) \urcorner$.*

Proof. (i) By induction on the length of s . The base case is trivial. Step case: suppose that $s \cdot m$ is a legal well-bracketed position.

- If m is an initial O-question then $?(\ulcorner s \cdot m \urcorner) = ?(m) = m = \ulcorner ?(s) \cdot m \urcorner = \ulcorner ?(s \cdot m) \urcorner$.
- If m is a non initial O-question then $s \cdot m^O = s' \cdot q^P \cdot s'' \cdot m^O$ where m is justified by q . We have $?(\ulcorner s \urcorner) = ?(\ulcorner s' \urcorner \cdot q \cdot m) = ?(\ulcorner s' \urcorner) \cdot q \cdot m$. If s' is not empty then its last move must be an O-move (by alternation), therefore by the induction hypothesis $?(\ulcorner s' \urcorner) = \ulcorner ?(s') \urcorner$. By the previous lemma, m 's justifier occurs in $?(s)$ therefore $?(s \cdot m) = ?(s') \cdot q^P \cdot u \cdot m^O$ for some sequence u and thus $\ulcorner ?(s \cdot m) \urcorner = \ulcorner ?(s') \urcorner \cdot q^P \cdot m^O$.
- If m is an O-answer then $s \cdot m = s' \cdot q^P \cdot s'' \cdot m^O$ where m is justified by q . We then have $?(\ulcorner s \cdot m \urcorner) = ?(\ulcorner s' \urcorner q a) = ?(\ulcorner s' \urcorner)$ and since s is well-bracketed, we have $?(s) = ?(s')$. The induction hypothesis permits us to conclude.
- If m is a P-question then $\ulcorner s \cdot m \urcorner = \ulcorner s \urcorner \cdot m$ and $?(\ulcorner s \cdot m \urcorner) = ?(\ulcorner s \urcorner) \cdot m$. Moreover $\ulcorner ?(s \cdot m) \urcorner = \ulcorner ?(s) \cdot m \urcorner = \ulcorner ?(s) \urcorner \cdot m$. By alternation if s is not empty it must end with an O-move so we can conclude using the induction hypothesis.

(ii) The argument is similar to (i). \square

Note that in (i) and (ii), it is important that s does not end with a P-answer. For instance consider the legal position

$$s = q_0^O \overset{\curvearrowright}{q_1^P} \overset{\curvearrowright}{q_2^O} \overset{\curvearrowright}{q_3^P} \overset{\curvearrowright}{q_4^O} a^P$$

ending with a P-answer. We have $\ulcorner ?(s) \urcorner = \ulcorner q_0 \cdot q_1 \cdot q_2 \cdot q_3 \urcorner = q_0 \cdot q_1 \cdot q_2 \cdot q_3$ but $?(\ulcorner s \urcorner) = ?(q_0 \cdot q_1 \cdot q_4 \cdot a) = q_0 \cdot q_1 \cdot q_4$.

By the previous remark and lemma we obtain the following corollary:

Corollary 2.3.1. *Let $s \cdot m$ be a legal well-bracketed position.*

- (i) *If m is a P-move then it points in $?(\ulcorner s \urcorner) = \ulcorner ?(s) \urcorner$;*
- (ii) *if m is a non initial O-move then it points in $?(\ulcorner s \urcorner) = \ulcorner ?(s) \urcorner$.*

Definition 2.3.10 (Order). Let $\langle M, \lambda, \vdash \rangle$ be a game. The **height** of a move m in M is the length of the longest chain of enabling moves $m \vdash m_2 \vdash \dots \vdash m_h$. The **order** of a move m written $\text{ord } m$ is defined as its height minus two. The order of a game $\langle M, \lambda, \vdash \rangle$ is defined as $\max_{m \in M} \text{ord } m$.

We make the assumption that each question move in the arena enables at least one answer move. Consequently, order-0 moves enable answer moves only.

Lemma 2.3.3 (Pointers are superfluous up to order 2). *Let A be an arena of order at most 2. Let s be a justified sequence of moves in the arena A satisfying alternation, visibility, well-openedness and well-bracketing. If s contains a single initial move then the pointers of the sequence s can be uniquely reconstructed from the underlying sequence of moves.*

Proof. Let A be an arena of order 2 at most and let s be a legal well-bracketed position in L_A . W.l.o.g. we can assume that the game A has a single initial move q_0 . Indeed, since s is well-opened, its first move m_0 is the only initial move in the sequence, thus m_0 is the root of some sub-arena A' of A . Hence s can be seen as a play on the game A' instead of A .

Since A is of order 2 at most, all the moves in s except q_0 are of order 1 at most. We prove by induction on the length of s that $?(s)$ corresponds to one of the cases 0, A, B, C, D shown on the table below, and that the pointers in s can be recovered uniquely. Let L denote the language $L = \{ pq \mid q_0 \vdash p \vdash q \wedge \text{ord } p = 1 \wedge \text{ord } q = 0 \}$.

Case	$\lambda_{OP}(m)$	$?(s) \in$	where...
0	O	$\{\epsilon\}$	
A	P	q_0	
B	O	$q_0 \cdot L^* \cdot p$	$q_0 \vdash p, \text{ord } p = 1$
C	P	$q_0 \cdot L^* \cdot pq$	$q_0 \vdash p \vdash q, \text{ord } p = 1, \text{ord } q = 0$
D	O	$q_0 \cdot L^* \cdot q$	$q_0 \vdash q, \text{ord } q = 0$

Base cases: If s is the empty play then there is no pointer to recover and s corresponds to case 0. If s is a singleton then it must be the initial question q_0 , so there is no pointer to recover. This corresponds to case A.

Step case: If $s = u \cdot m$ for some non empty legal well-bracketed position u and move $m \in M_A$ then by the induction hypothesis the pointers in u can all be recovered and u corresponds to one of the cases 0, A, B, C or D. We proceed by case analysis:

case 0 $?(u) = \epsilon$. By corollary 2.3.1, m points in $\ulcorner ?(u) \urcorner = \epsilon$. Hence this case is impossible.

case A $?(u) = q_0$ and the last move m is played by P. By corollary 2.3.1, m points to q_0 . If m is an answer to the initial question q_0 then s is a complete play and $?(s) = \epsilon$, which corresponds to case 0. If m is a first order question then $?(s) = q_0 p$ and it is O's turn to play after s therefore s falls into category B. If m is an order 0 question then s falls into category D.

case B $?(u) \in q_0 \cdot L^* \cdot p$ where $\text{ord } p = 1$ and m is an O-move. By corollary 2.3.1, m points in $\ulcorner ?(u) \urcorner = q_0 p$. Since m is an O-move it can only point to p . If m is an answer to p then $?(s) = ?(u \cdot m) \in q_0 \cdot L^*$ which is covered by case A and C. If m is an order 0 question pointing to p then we have $?(s) = ?(u) \cdot m \in q_0 \cdot L^* \cdot pm$ and s falls into category C.

case C $?(u) \in q_0 \cdot L^* \cdot pq$ where $\text{ord } p = 1, \text{ord } q = 0$, q_0 justifies p , p justifies q and m is played by P.

Suppose that m is an answer, then the well-bracketing condition imposes q to be answered first. The move m therefore points to q and we have $?(s) = ?(u \cdot m) \in q_0 \cdot L^* \cdot p$. This corresponds to case B.

Suppose that m is a question. m is a P-move therefore is cannot be justified by p . It cannot be justified by q either because q is an order 0 question and therefore enables answer moves only. Similarly m is not justified by any move in L^* . Hence m must point to the initial question q_0 . There are two sub-cases, either m is an order 0 move and then s falls into category D or m is an order 1 move and s falls into category B.

case D $?(u) \in q_0 \cdot L^* \cdot q$ where $\text{ord } q = 0$ and m is played by O.

Again by corollary 2.3.1, m points in $\ulcorner ?(u) \urcorner = q_0 q$. Since m is a P-move it can only point to q . Since q is of order 0, it only enables answer moves therefore m is an answer to q . Hence $?(s) = ?(u \cdot m) \in q_0 \cdot L^*$ and s falls either into category A or C. \square

Hence for order-2 games, plays are entirely determined by underlying pointer-less sequence of moves. At order 3, however, eliminating pointers causes ambiguities. Take for instance the

game $((\mathbb{N}^1 \rightarrow \mathbb{N}^2) \rightarrow \mathbb{N}^3) \rightarrow \mathbb{N}^4$ and sequence of moves $s = q^4 q^3 q^2 q^3 q^2 q^1$, where the superscripts indicate the component of the game in which each move is played. What are the valid plays whose underlying sequence of moves is s ? By the visibility condition, the pointers of the first five moves are uniquely determined:

$$s = q^4 \overset{\curvearrowright}{q^3} \overset{\curvearrowright}{q^2} \overset{\curvearrowright}{q^3} \overset{\curvearrowright}{q^2} q^1 .$$

For the last move, however, there is an ambiguity: its justifier can be any of the two occurrences of q^2 . The visibility condition does not eliminate this ambiguity since both occurrences of q^2 appear in the P-view $\lceil s \rceil = s$. These two possibilities correspond to two different strategies for the Proponent.

2.3.4 Categorical interpretation

This section recalls briefly the categorical interpretation of games [McC96a, HO00, AMJ94]. We consider the category [Cro93] \mathcal{G} whose objects are games and morphisms are strategies. A morphism from A to B is a strategy on the game $A \multimap B$. Three other sub-categories of \mathcal{G} are considered, each of them corresponds to some restriction on strategies: \mathcal{G}_i is the sub-category of \mathcal{G} whose morphisms are the innocent strategies, \mathcal{G}_b has only the well-bracketed strategies and \mathcal{G}_{ib} has the innocent and well-bracketed strategies.

Proposition 2.3.1. \mathcal{G} , \mathcal{G}_i , \mathcal{G}_b and \mathcal{G}_{ib} are categories.

In particular this means that composition of strategies is well-defined, associative, has a unit (the copy-cat strategy), preserves innocence and well-bracketedness [HO00, AMJ94].

2.3.4.1 Monoidal structure

We have already defined the tensor product on games in Section 2.3.2.3. We now define the corresponding transformation on morphisms. Given two strategies $\sigma : A \multimap B$ and $\tau : C \multimap D$ the strategy $\sigma \otimes \tau : (A \otimes C) \multimap (B \otimes D)$ is defined by:

$$\sigma \otimes \tau = \{s \in L_{A \otimes C \multimap B \otimes D} \mid s \upharpoonright A, B \in \sigma \wedge s \upharpoonright C, D \in \tau\} .$$

It can be shown that the tensor product is associative, commutative and has $I = \langle \emptyset, \emptyset, \emptyset, \{\epsilon\} \rangle$ as identity. Hence the game category \mathcal{G} is a symmetric monoidal category. Moreover \mathcal{G}_i and \mathcal{G}_b are sub-symmetric monoidal categories of \mathcal{G} , and \mathcal{G}_{ib} is a sub-symmetric monoidal category of \mathcal{G}_i , \mathcal{G}_b and \mathcal{G} .

2.3.4.2 Closed structure

Given the games A , B and C , we can transform strategies on $A \otimes B \multimap C$ to strategies on $A \multimap (B \multimap C)$ by retagging the moves to the appropriate arenas. This transformation defines an isomorphism written Λ_B and called *currying*. Therefore the hom-set $\mathcal{G}(A \otimes B, C)$ is isomorphic to the hom-set $\mathcal{G}(A, B \multimap C)$ which makes \mathcal{G} an autonomous (*i.e.*, symmetric monoidal closed) category.

We write $ev_{A,B} : (A \multimap B) \otimes A \rightarrow B$ to denote the *evaluation strategy* obtained by uncurrying the identity map on $A \rightarrow B$. $ev_{A,B}$ is in fact the copycat strategy for the game $(A \multimap B) \otimes A \rightarrow B$.

\mathcal{G}_i and \mathcal{G}_b are sub-autonomous categories of \mathcal{G} , and \mathcal{G}_{ib} is a sub-autonomous category of \mathcal{G}_i , \mathcal{G}_b and \mathcal{G} .

2.3.4.3 Cartesian product

The cartesian product defined in Section 2.3.2.3 is indeed a cartesian product in the category \mathcal{G} , \mathcal{G}_i , \mathcal{G}_b and \mathcal{G}_{ib} . The projections $\pi_1 : A \& B \rightarrow A$ and $\pi_2 : A \& B \rightarrow B$ are given by the obvious

copy-cat strategies. Given two category morphisms $\sigma : C \rightarrow A$ and $\tau : C \rightarrow B$, the *pairing* morphism $\langle \sigma, \tau \rangle : C \rightarrow A \& B$ is given by:

$$\begin{aligned} \langle \sigma, \tau \rangle &= \{s \in L_{C \rightarrow A \& B} \mid s \upharpoonright C, A \in \sigma \wedge s \upharpoonright B = \epsilon\} \\ &\cup \{s \in L_{C \rightarrow A \& B} \mid s \upharpoonright C, B \in \tau \wedge s \upharpoonright A = \epsilon\} . \end{aligned}$$

2.3.4.4 Cartesian closed structure

To obtain a cartesian closed category: we need to define a *terminal* object and the *exponential* construct $A \Rightarrow B$ for any two games A and B .

The empty game $\mathbf{1}$ is terminal. For any game A the *exponential* game $!A$ is given by:

$$\begin{aligned} M_{!A} &= M_A \\ \lambda_{!A} &= \lambda_A \\ \vdash_{!A} &= \vdash_A \\ P_{!A} &= \{s \in L_{!A} \mid \text{for each initial move } m, s \upharpoonright m \in P_A\} . \end{aligned}$$

Think of it as the multi-threaded version of the game A in which a new copy the game can be spawned at any time. Plays of $!A$ are thus an interleaving of plays of A . We have the following identities:

$$\begin{aligned} !(A \& B) &= !A \otimes !B \\ \mathbf{1} &= !\mathbf{1} . \end{aligned}$$

A game A is said to be *well-opened* if for any position $s \in P_A$ the only initial move in s is the first one. In a well-opened game, plays contain a single “thread” of moves. Given a strategy on a well-opened game, one can turn it into a “multi-threaded” strategy using the *promotion* operator:

Definition 2.3.11 (Promotion). Consider a well-opened game B . Given a strategy on $!A \multimap B$, its *promotion* $\sigma^\dagger : !A \multimap !B$ is the strategy which plays several copies of σ . Formally:

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial } m, s \upharpoonright m \in \sigma\} .$$

It can be shown that promotion is a well-defined strategy and that it preserves innocence and well-bracketing. We now introduce the category of well-opened games:

Definition 2.3.12 (Category of well-opened games). The category \mathcal{C} of well-opened games, also called the *co-Kleisli category* of \mathcal{G} , is defined as follows:

1. The objects are the well-opened games,
2. a morphism $\sigma : A \rightarrow B$ is a strategy for the game $!A \multimap B$,
3. the identity map for A is the copy-cat strategy on $!A \multimap A$ (which is well-defined for well-opened games). It is called dereliction, denoted by \mathbf{der}_A and defined formally by:

$$\mathbf{der}_A = \{s \in P_{!A \multimap A}^{\text{even}} \mid \forall t \leq^{\text{even}} s . t \upharpoonright !A = t \upharpoonright A\},$$

4. composition of morphisms $\sigma : !A \multimap B$ and $\tau : !B \multimap C$ denoted by $\sigma \circ \tau : !A \multimap C$ is defined as $\sigma^\dagger ; \tau$.

\mathcal{C} is a well-defined category and has three sub-categories $\mathcal{C}_i, \mathcal{C}_b, \mathcal{C}_{ib}$ corresponding respectively to sub-category of innocent, well-bracketed, and innocent and well-bracketed strategies.

The category \mathcal{C} has a terminal object $\mathbf{1}$, and for any two games A and B , a product $A \& B$ and an exponential $A \Rightarrow B$ defined as $!A \multimap B$. The hom-sets $\mathcal{C}(A \& B, C)$ and $\mathcal{C}(A, !B \multimap C)$ are isomorphic. Indeed:

$$\begin{aligned} \mathcal{C}(A \& B, C) &= \mathcal{G}(! (A \& B), C) \\ &= \mathcal{G}(!A \otimes !B, C) \\ &\cong \mathcal{G}(!A, !B \multimap C) \quad (\mathcal{G} \text{ is a closed monoidal category}) \\ &= \mathcal{C}(A, !B \multimap C) . \end{aligned}$$



Hence \mathcal{C} is a cartesian closed category. Furthermore \mathcal{C}_i and \mathcal{C}_b are sub-cartesian closed categories of \mathcal{C} and \mathcal{C}_{ib} is a sub-cartesian closed category of each of \mathcal{C} , \mathcal{C}_i and \mathcal{C}_b .

2.3.4.5 Order enrichment

Strategies can be ordered using the inclusion ordering. Under this ordering, the set of strategies on a given game A is a pointed directed complete partial order: the least upper bound is given by the set-theoretic union and the least element is the empty strategy $\{\epsilon\}$.

Moreover all the operators on strategies that we have defined so far (composition, tensor product, ...) are continuous. Hence the categories \mathcal{C} and \mathcal{G} are cpo-enriched.

2.3.4.6 Intrinsic preorder

Let Σ denote the *Sierpinski game* with a single question q and single answer a . There are only two strategies on Σ : $\perp = \{\epsilon\}$ and $\top = \{\epsilon, qa\}$ which are both innocent and well-bracketed. For any object A , the ***intrinsic preorder*** \lesssim_A on the set of strategies on the game A is defined by:

$$\sigma \lesssim_A \tau \iff \forall \alpha : A \rightarrow \Sigma. \sigma \circ \tau = \top \implies \tau \circ \alpha = \top .$$

This indeed defines a preorder [AMJ94]. The ***quotiented category*** \mathcal{C}/\lesssim is defined as follows. The objects of \mathcal{C}/\lesssim are those of \mathcal{C} , and the morphisms are the equivalence classes of morphism in \mathcal{C} modulo the equivalence relation induced by \lesssim .

We will consider the quotiented categories $\mathcal{C}_\$/\lesssim_\$$ where $\$$ ranges in $\{i, b, ib\}$. The full abstraction of the game semantic model of PCF will be stated in the quotiented category $\mathcal{C}_{ib}/\lesssim_{ib}$ rather than \mathcal{C}_{ib} .

2.3.5 The fully abstract game model of PCF

This section presents the fully abstract Hyland-Ong game model of PCF [HO00]. As we have seen in Section 2.3.2, games and strategies form a cartesian closed category, therefore games can model the simply typed lambda-calculus. We now make this connection explicit by associating a strategy to any given lambda-term. We then extend the model to PCF and IA.

2.3.5.1 Simply typed lambda calculus fragment

In the games that we are considering, the Opponent (O) represents the environment and the Proponent (P) represents the lambda term. Opponent opens the game by asking a question such as “What is the output of the function?”. Then the proponent may ask further information such that “What is the input of the function?”. O can provide P with an answer—which is the value of the input—or can pursue with another question. The dialog goes on until O gets the answer to his initial question.

O represents the environment, he is responsible for proving input values while P plays from the term’s point of view: he is responsible for performing the computation and returning the output to O. P plays according to the strategy that is associated to the lambda-term being modeled.

We recall that in the cartesian closed category \mathcal{C} , the objects are the games and the morphisms are the strategies. The interpretation of a simple type A is denoted $\llbracket A \rrbracket$. A context $\Gamma = x_1 :$

$A_1, \dots, x_n : A_n$ will be mapped to the game $\llbracket \Gamma \rrbracket = \llbracket A_1 \rrbracket \times \dots \times \llbracket A_n \rrbracket$ and a term $\Gamma \vdash M : A$ will be modeled by a strategy on the game $\llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$. The terminal object $\mathbf{1}$ of the cartesian closed category \mathcal{C} models the empty context: $\llbracket \Gamma \rrbracket = \mathbf{1}$.

The base type **exp** is interpreted by the flat game \mathbb{N} over the natural number. This game has only one question: the initial O-question; P can then answer by playing a natural number $i \in \mathbb{N}$. There exist only two kinds of strategies for this game:

- The empty strategy where P never answer the initial question. This corresponds to a non terminating computation;
- The strategies where P answers by playing a number n . This models a numerical constant of the language.

Given the interpretation of base types, the interpretation of $A \rightarrow B$ is then given by the exponential object of the cartesian closed category \mathcal{C} :

$$\llbracket A \rightarrow B \rrbracket = \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket = !\llbracket A \rrbracket \multimap \llbracket B \rrbracket .$$

The strategy denotation of a term is defined inductively on the structure of the term:

- Variables are interpreted by projection:

$$\llbracket x_1 : A_1, \dots, x_n : A_n \vdash x_i : A_i \rrbracket = \pi_i : \llbracket A_1 \rrbracket \times \dots \times \llbracket A_i \rrbracket \times \dots \times \llbracket A_n \rrbracket \rightarrow \llbracket A_i \rrbracket .$$

- The abstraction $\Gamma \vdash \lambda x^A.M : A \rightarrow B$ is modeled by a strategy on the arena $\llbracket \Gamma \rrbracket \rightarrow (\llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket)$. This strategy is obtained by currying:

$$\llbracket \Gamma \vdash \lambda x^A.M : A \rightarrow B \rrbracket = \Lambda(\llbracket \Gamma, x : A \vdash M : B \rrbracket) .$$

- The application $\Gamma \vdash MN$ is modeled using the evaluation map $ev_{A,B} : (A \Rightarrow B) \times A \rightarrow B$:

$$\llbracket \Gamma \vdash MN \rrbracket = \langle \llbracket \Gamma \vdash M, \Gamma \vdash N \rrbracket \rangle ; ev_{A,B} .$$

Example 2.3.4 (Kierstead terms). In Section 2.3.3 we have shown that there are two different strategies on the game $((\mathbb{N}^1 \rightarrow \mathbb{N}^2) \rightarrow \mathbb{N}^3) \rightarrow \mathbb{N}^4$ containing a play whose underlying sequence of move is $q^4 q^3 q^2 q^3 q^2 q^1$ but whose justification pointers differ.

These two strategies are precisely the denotation of the *Kierstead terms* defined as follows:

$$\begin{aligned} M_1 &= \lambda f.f(\lambda x.f(\lambda y.y)) : ((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\ M_2 &= \lambda f.f(\lambda x.f(\lambda y.x)) : ((\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}) \rightarrow \mathbb{N} . \end{aligned}$$

Suppose that q^1 is justified by the first occurrence of q^2 then it means that the proponent is requesting the value of the variable x bound in the subterm $\lambda x.f(\lambda y.y)$. If P needs to know the value of x , this is because P is in fact following the strategy of the subterm $\lambda y.x$. This corresponds to a play of the strategy $\llbracket M_2 \rrbracket$. If q^1 is justified by the second occurrence of q^2 then it is a play of $\llbracket M_1 \rrbracket$.

2.3.5.2 PCF fragment

We now show how to model PCF constructs. In the following, we tag each sub-arena in a game so that it is possible to distinguish identical arenas occurring in different components of the game. We also tag moves (in the exponent) so that it is possible to identify the arena component in which the move belongs. We will omit the pointers in the play when there is no ambiguity.

The successor arithmetic operator is modeled by the following strategy on the arena $\mathbb{N}^1 \Rightarrow \mathbb{N}^0$:

$$\llbracket \text{succ} \rrbracket = \text{Pref}^{\text{even}} \{ q^0 \cdot q^1 \cdot n^1 \cdot (n+1)^0 \mid n \in \mathbb{N} \} .$$

where $\text{Pref}^{\text{even}} X$ denotes the set consisting of the prefixes of even length of plays of X .

The predecessor arithmetic operator is denoted by the strategy

$$\llbracket \text{pred} \rrbracket = \text{Pref}^{\text{even}}(\{q^0 \cdot q^1 \cdot n^1 \cdot (n-1)^0 \mid n > 0\} \cup \{q^0 \cdot q^1 \cdot 0^1 \cdot 0^0\}) .$$

Then given a term $\Gamma \vdash \text{succ } M : \text{exp}$ we can define:

$$\llbracket \Gamma \vdash \text{succ } M : \text{exp} \rrbracket = \llbracket \Gamma \vdash M \rrbracket ; \llbracket \text{succ} \rrbracket$$

$$\llbracket \Gamma \vdash \text{pred } M : \text{exp} \rrbracket = \llbracket \Gamma \vdash M \rrbracket ; \llbracket \text{pred} \rrbracket .$$

The conditional operator is denoted by the following strategy on the arena $\mathbb{N}^3 \times \mathbb{N}^2 \times \mathbb{N}^1 \Rightarrow \mathbb{N}^0$:


$$\llbracket \text{cond} \rrbracket = \text{Pref}^{\text{even}}\{q^0 \cdot q^3 \cdot 0 \cdot q^2 \cdot n^2 \cdot n^0 \mid n \in \mathbb{N}\} \cup \text{Pref}^{\text{even}}\{q^0 \cdot q^3 \cdot m \cdot q^2 \cdot n^2 \cdot n^0 \mid m > 0, n \in \mathbb{N}\} .$$

Given a term $\Gamma \vdash \text{cond } M \ N_1 \ N_2$ we define:

$$\llbracket \Gamma \vdash \text{cond } M \ N_1 \ N_2 \rrbracket = \langle \llbracket \Gamma \vdash M \rrbracket, \llbracket \Gamma \vdash N_1 \rrbracket, \llbracket \Gamma \vdash N_2 \rrbracket \rangle ; \llbracket \text{cond} \rrbracket .$$

The interpretation of the Y combinator is a bit more complicated. Consider the term $\Gamma \vdash M : A \rightarrow A$, its semantics f is a strategy on $\llbracket \Gamma \rrbracket \times \llbracket A \rrbracket \rightarrow \llbracket A \rrbracket$. We define the chain g_n of strategies on the arena $\llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$ as follows:

$$\begin{aligned} g_0 &= \perp \\ g_{n+1} &= F(g_n) = \langle id_{\llbracket \Gamma \rrbracket}, g_n \rangle ; f \end{aligned}$$

where \perp denotes the empty strategy $\{\epsilon\}$. It is easy to see that the g_n forms a chain. The denotation $\llbracket YM \rrbracket$ is defined as the least upper bound of the chain g_n which is also the least fixed point of F . Its existence is guaranteed by the fact that the category of games is cpo-enriched .

Since all the strategies that we have given are innocent and well-bracketed, the game model of PCF can be interpreted in any of the four categories \mathcal{C} , \mathcal{C}_i , \mathcal{C}_b , \mathcal{C}_{ib} . The category \mathcal{C}_{ib} is referred as the *intentional game model* of PCF.

2.3.5.3 Observational preorder

A context denoted $C[-]$ is a term containing a hole denoted by $-$. If $C[-]$ is a context then $C[M]$ denotes the term obtained after replacing the hole by the term M . $C[M]$ is well-formed provided that M has the appropriate type. Note that this substitution is done capture-permitting, as opposed to the capture-avoiding substitution used to contract beta-redexes in the lambda calculus.

Definition 2.3.13. The *observational preorder* is a relation \sqsubseteq on terms defined as follows: for any two closed terms M and N of the same type,

$$M \sqsubseteq N \iff \text{for all context } C[-] \text{ such that } C[M] \text{ and } C[N] \text{ are well-formed closed PCF term of type } \text{exp}, C[M] \Downarrow \text{ implies } C[N] \Downarrow .$$

The reflexive closure of \sqsubseteq , denoted \cong , is called the *observational equivalence* relation.

The intuition behind this definition is that two terms are observationally equivalent if there is no context that distinguishes them, and therefore they can be safely interchanged in any program context.

2.3.5.4 Soundness

We say that the model is *sound for evaluation* if the denotation of a term is preserved by the evaluation relation \Downarrow of the big-step semantics of the language: for any term M and value V we have:

$$M \Downarrow V \implies \llbracket M \rrbracket = \llbracket V \rrbracket .$$

Lemma 2.3.4 ([AM98b]). *The game model of PCF is sound for evaluation.*

Definition 2.3.14 (Computable terms).

- A closed term $\vdash M : B$ of base type is computable if $\llbracket M \rrbracket \neq \perp$ implies $M \Downarrow$.
- A higher-order closed term $\vdash M : A \rightarrow B$ is computable if MN is computable for any computable closed term $\vdash N : A$.
- An open term $x_1 : A_1, \dots, x_n : A_n \vdash M : A \rightarrow B$ is computable if $\vdash M[N_1/x_1, \dots, N_n/x_n]$ is computable for all computable closed terms $N_1 : A_1, \dots, N_n : A_n$.

A model is **computationally adequate** if all terms are computable.

Lemma 2.3.5 ([AM98b]). *The game model of PCF is computationally adequate.*

A model of a programming language is said to be **sound** if whenever the denotation of two programs are equal then the two programs are observationally equivalent; formally for any closed terms M and N of the same type we have:

$$\llbracket M \rrbracket = \llbracket N \rrbracket \implies M \cong N.$$


The model is said to be **inequationally sound** if the following stronger condition holds

$$\llbracket M \rrbracket \subseteq \llbracket N \rrbracket \implies M \sqsubseteq N.$$

Soundness is the minimum one can require for a model of programming language: it tells us that we can reason about terms by manipulating objects in the denotational model.

Inequational soundness of PCF follows from the last two lemmas:

Proposition 2.3.2. *The game model of PCF is inequationally sound.*

Proof. Take two  closed PCF terms M and N . Suppose that $\llbracket M \rrbracket \subseteq \llbracket N \rrbracket$ then by compositionality of the model we have $\llbracket C[M] \rrbracket \subseteq \llbracket C[N] \rrbracket$. Suppose that $C[M] \Downarrow$ for some context $C[-]$ then by soundness (Lemma 2.3.4) we have $\llbracket C[M] \rrbracket \neq \perp$, which implies $\llbracket C[N] \rrbracket \neq \perp$. The adequacy of the model (Lemma 2.3.5) then gives us $C[N] \Downarrow$. Hence $M \sqsubseteq N$. \square

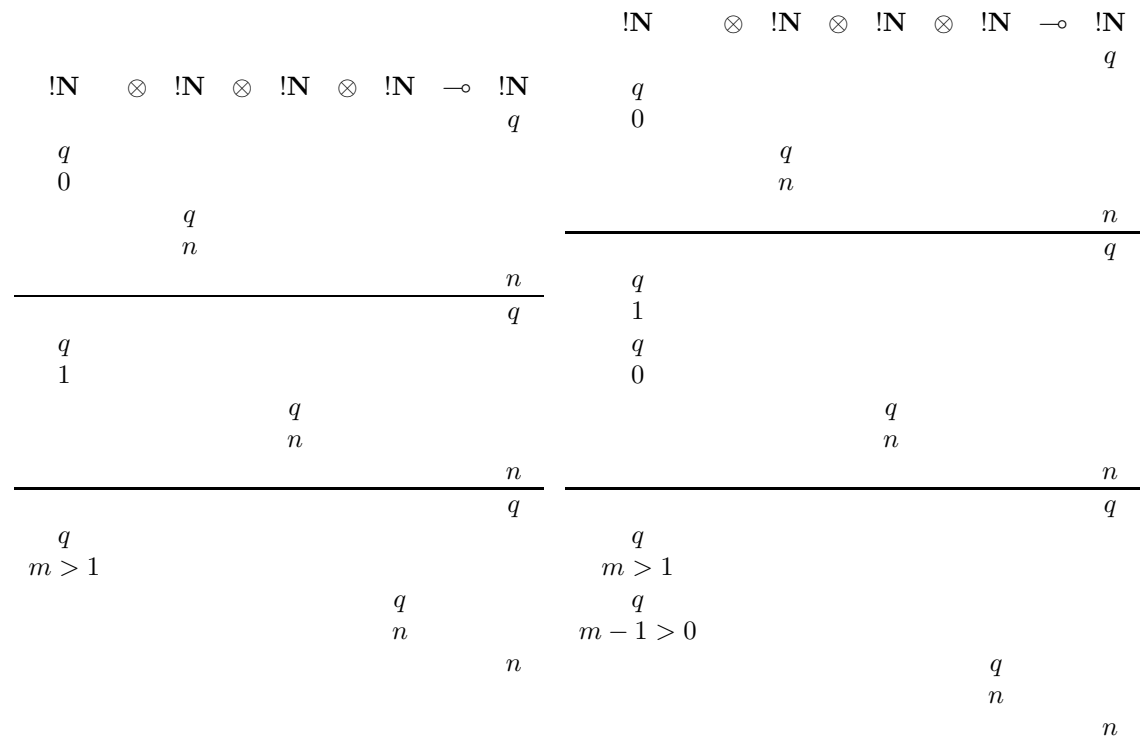
2.3.5.5 Definability

We now work in the category \mathcal{C}_{ib} of innocent and well-bracketed strategies. The *definability* property is the key to the full-abstraction result. It says that every compact element of the model is the denotation of some term. In \mathcal{C}_{ib} , the **compact morphisms** are the innocent strategies with finite view-function. Due to its economical syntax, PCF does not verify the definability result: there are strategies that are not the denotation of any term in PCF. For instance consider the *ternary conditional* strategy acting as follows: it tests the value of its first parameter, if it is equal to 0 or 1 then it returns the value of the second or third parameter respectively, otherwise it returns the value of the fourth parameter. This is illustrated in the left diagram of Fig. 2.3.5.5. Such computation can be operationally simulated in PCF by the term $T_3 = \text{cond } M \ N_1(\text{cond } (\text{pred } M) \ N_2 \ N_3)$. The term T_3 , however, is not denoted by the ternary conditional strategy. Its denotation is instead given by the right diagram on Fig. 2.3.5.5.

In PCF_c , however, the ternary conditional strategy is definable by the term case_3 . In fact, the definability result holds precisely for PCF_c :

Proposition 2.3.3 (Definability). *Let A be a PCF type and σ be a compact innocent and well-bracketed strategy on A . There exists a PCF_c term M such that $\llbracket M \rrbracket = \sigma$.*

The definability only holds for PCF_c but crucially, PCF_c is a conservative extension of PCF: if M and N are terms such that for any PCF-context $C[-]$, $C[M] \Downarrow \implies C[N] \Downarrow$ then the same is true for any PCF_c -context. (This is because the case_k constructs can all be simulated by PCF terms with the same operational semantics.) This property suffices to prove full abstraction of PCF.

Figure 2.1: Strategy denotation of $case_3$ (left) and T_3 (right).

2.3.5.6 Full abstraction

The converse of soundness is called *completeness*: a model is **complete** if:

$$M \cong N \implies \llbracket M \rrbracket = \llbracket N \rrbracket .$$

Further, if the stronger relation

$$M \sqsubset N \implies \llbracket M \rrbracket \subseteq \llbracket N \rrbracket$$

holds then the model is said to be **inequationally complete**.

A model is **fully abstract** if it is both sound and complete, and **inequationally fully abstract** if it is inequationally sound and inequationally complete.

Full abstraction of PCF cannot be stated directly in the category \mathcal{C}_{ib} . Instead we need to consider the quotiented category $\mathcal{C}_{ib}/\lesssim_{ib}$. But first we need to make sure that $\mathcal{C}_{ib}/\lesssim_{ib}$ is a model of PCF. $\mathcal{C}_{ib}/\lesssim_{ib}$ is a poset-enriched cartesian closed category. The denotation of the basic types and constants of PCF can be transposed from \mathcal{C}_{ib} to $\mathcal{C}_{ib}/\lesssim_{ib}$. Although it is not known whether $\mathcal{C}_{ib}/\lesssim_{ib}$ is enriched over the category of CPOs, it can be proved that it verifies a condition called *rationality* [AMJ94] and this suffices to ensure that $\mathcal{C}_{ib}/\lesssim_{ib}$ is indeed a model of PCF. This category will be referred as the **extensional game model** of PCF. The full abstraction of the game model then follows from Proposition 2.3.2 and 2.3.3:

Theorem 2.3.2 (Full abstraction [AMJ94, HO00, Nic94]). *Let M and N be two closed PCF terms.*

$$\llbracket M \rrbracket \lesssim_{ib} \llbracket N \rrbracket \iff M \sqsubset N$$

where \lesssim_{ib} denotes the intrinsic preorder of the category \mathcal{C}_{ib} .

2.3.6 The fully abstract game model of Idealized Algol

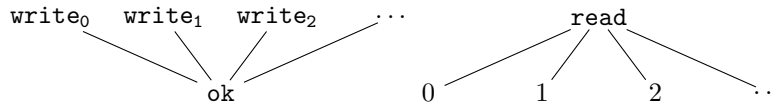
We now describe the fully abstract game model of IA [AM99].

All the strategies used to model PCF are well-bracketed and innocent. To obtain a model of IA, however we need to introduce strategies that are not innocent. This is necessary to model the memory cell variable created with the **new** operator. The intuition is that a cell needs to remember the last value which was written in it in order to be able to return it when it is read, and this can only be done by looking at the whole history of moves, not only those present in the P-view. We therefore restrict our attention to the categories \mathcal{C} and \mathcal{C}_b .

Base types

The type **com** is modeled by the flat game with a single initial question **run** and a single answer **done**. The idea is that O can request the execution of a command by playing **run**, P then executes the command and if it terminates, acknowledges it by playing **done**.

The variable type **var** is modeled by the game $\mathbf{com}^N \times \mathbf{exp}$ illustrated below:



Modelling the constants

- The constant **skip** is interpreted by the strategy $\{\epsilon, \mathbf{run} \cdot \mathbf{done}\}$.

- Sequential composition seq_{exp} is interpreted by the following strategy:

$$\begin{array}{c}
 !\text{com} \otimes !\text{exp} \xrightarrow{\text{seq}_{\text{exp}}} \text{exp} \\
 \text{run} \\
 \text{done} \\
 q \\
 n \\
 n
 \end{array}$$

- Assignment assign and dereferencing deref are denoted by the following strategies:

$$\begin{array}{c}
 !\text{var} \otimes !\text{exp} \xrightarrow{\text{assign}} \text{com} \\
 q \\
 n \\
 \text{write}_n \\
 \text{ok} \\
 \text{done}
 \end{array}
 \quad
 \begin{array}{c}
 !\text{var} \xrightarrow{\text{deref}} \text{exp} \\
 q \\
 \text{read} \\
 n \\
 n
 \end{array}$$

- mkvar is modeled by the paired strategy $\langle \text{mkvar}_{\text{acc}}, \text{mkvar}_{\text{exp}} \rangle$ where $\text{mkvar}_{\text{acc}}$ and $\text{mkvar}_{\text{exp}}$ are the following strategies:

$$\begin{array}{c}
 !(\text{!exp} \multimap \text{com}) \otimes !\text{exp} \xrightarrow{\text{mkvar}_{\text{acc}}} \text{com}^\omega \\
 \text{write}_n \\
 \text{run} \\
 q \\
 n \\
 \text{done}
 \end{array}
 \quad
 \begin{array}{c}
 !(\text{!exp} \multimap \text{com}) \otimes !\text{exp} \xrightarrow{\text{mkvar}_{\text{exp}}} \text{exp} \\
 q \\
 n \\
 \text{read} \\
 n
 \end{array}$$

- Block-allocated variable (**new**): The strategies introduced until now are all innocent. In order to model the **new** operator, it is necessary to introduce non-innocent strategies, also called *knowing strategies*. We call **memory-cell strategy** the knowing well-bracketed strategy written $\text{cell} : I \multimap !\text{var}$ behaving as follows: it responds to **write** with **ok** and responds to **read** with the last value written or 0 if no value has yet been written. The denotation of a term $\Gamma \vdash \text{new } x \text{ in } M : A$ is then defined as the strategy:

$$\llbracket \Gamma \vdash \text{new } x \text{ in } M : A \rrbracket = (id_{\llbracket \Gamma \rrbracket} \otimes \text{cell}) ; \llbracket \Gamma, x : \text{var} \vdash M : A \rrbracket : !\Gamma \multimap \text{com} .$$

Full abstraction

The inequational soundness result can also be proved for IA. Proving soundness of the evaluation requires a bit more work than in the PCF case because the store needs to be made explicit. Also, one needs to define an appropriate notion of *computable term* that takes into account the presence of stores in the evaluation semantics. It is also possible to prove that the model is computational adequate. This implies:

Proposition 2.3.4 (Abramksy and McCusker [AMJ94]). *The game model of IA is inequationally sound.*

A result called the Innocent Factorization Theorem [AM97], shows that the strategies in \mathcal{G}_b can all be obtained by composing the non-innocent strategy cell with an innocent strategy. The strategy cell can therefore be viewed as a generic non-innocent strategy. Using this factorization argument, it is possible to prove the definability result:

Proposition 2.3.5 (Definability). *For any compact well-bracketed strategy σ on a game A denoting a IA type, there exists an IA-term M such that $\llbracket M \rrbracket = \sigma$.*

Full abstraction for the model \mathcal{C}_b is then a consequence of inequational soundness and definability:

Theorem 2.3.3 (Full abstraction). *Let M and N be two closed IA-terms.*

$$\llbracket M \rrbracket \lesssim_b \llbracket N \rrbracket \iff M \sqsubseteq N$$

where \lesssim_b denotes the intrinsic preorder of the category \mathcal{C}_b .

2.3.7 Algorithmic game semantics

Game semantics has proved to be a very successful paradigm in fundamental computer science. Following the resolution of the full abstraction problem for PCF, game semantics was subsequently used to obtain fully abstract models of a variety of programming languages. More recently, game semantics has emerged as a new approach to program verification and program analysis. Ghica and McCusker identified a fragment of Idealized Algol for which the game denotation of programs can be expressed using regular expressions; Consequently, the observational equivalence problem for this fragment is decidable [GM00, GM03]. This development opened up a new branch of research called *Algorithmic game semantics* which has interesting applications in program verification [AGOM03, DGL05]. This section gives a quick overview of some important results in algorithmic game semantics.

2.3.7.1 Effective presentability

The starting point of Algorithmic game semantics is a result shown by Abramsky and McCusker called the Characterization Theorem [AM97, Theorem 25]. We say that a play is *complete* if it is maximal and all question have been answered. One can show that for any IA type T , the complete plays on the game $\llbracket T \rrbracket$ are precisely those in which the initial question has been answered. A game verifying this condition is said to be *simple* [AM97]. The characterization theorem can then be stated as follows:

Theorem 2.3.4 (Characterization Theorem for Simple Game (Abramsky, McCusker [AM97])). *Let σ and τ be strategies on a simple game A then:*

$$\sigma \leq \tau \iff \text{comp}(\sigma) \subseteq \text{comp}(\tau) .$$

Thus in the game model of Idealized Algol, observational equivalence is characterized by equality of the set of complete plays.

This result implies that the fully abstract model of Idealized Algol is *effectively presentable* [Loa98b] which means that the denotation of a term can be computed by a Turing Machine. The proof crucially relies on the presence of imperative features in IA. Indeed, Loader has shown that even on compact strategies, observation equivalence of PCF is undecidable [Loa01]. This implies that there is not fully abstract model of PCF that is effectively representable.

Algorithmic game semantics is concerned with deriving decision procedures for the observational equivalence problem for various fragments of IA. This problem can be stated as follows: *Given two β -normal forms M and N in a given fragment of IA, does $M \cong N$ hold?* By the Characterization Theorem 2.3.4, this problem reduces to comparing the set of complete plays of two given terms. Observational equivalence is undecidable in the general case, but it becomes decidable when restricted to some lower-order fragments of IA. This question has now been fully investigated and there is now a full classification of decidability results for IA's fragments.

2.3.7.2 The order-2 fragment of IA

Ghica and McCusker were the first to show that the observational equivalence problem becomes decidable when restricting the language IA to some finitary fragment. They showed that for the second-order finitary fragment of Idealized Algol, written IA_2 , the set of complete plays of the strategy denotation can be expressed as an extended regular expression [GM00]:

Lemma 2.3.6 (Ghica and McCusker, [GM00]). *For any IA_2 -term $\Gamma \vdash M : T$, the set of complete plays of $\llbracket \Gamma \vdash M : T \rrbracket$ is regular.*

Since equivalence of regular expression is decidable with complexity PSPACE, by the Characterization Theorem this result gives a decision procedure for observational equivalence of IA_2 -terms. In the same paper they show that the same result holds for the $IA_2 + \text{while}$ fragment. At order 2, this results cannot be extend further as Ong showed that that observational equivalence is undecidable for $IA_2 + Y_1$ [Ong02].

2.3.7.3 Other fragments of IA

The result by Ghica and McCusker opened up the field of *Algorithmic game semantics*. Other finitary fragments were subsequently considered. Ong considered the order-3 finitary fragment, denoted IA_3 . He showed that the set of complete plays is a context-free language, thus observational equivalence reduces to the *Deterministic Push-down Automata Equivalence* (DPDA) [Ong02]. This problem was shown to be decidable [Sén01] but its complexity is still unknown. We only know that it is primitive recursive [Sti02].

Even for $IA_3 + \text{while}$ (the fragment obtained by throwing in iteration), the problem remains decidable. Moreover the problem lies in EXPTIME [MW05]. For the fragments $IA_i + Y_0$ for $i = 1, 2, 3$, observational equivalence is as difficult as DPDA equivalence (*i.e.*, there is a reduction in both directions) [MOW05]. Finally, Murawski showed that the problem becomes undecidable beyond order 3 (IA_i with $i \geq 4$) [Mur03].

The complete classification of complexity results for IA is recapitulated in Table 2.6. Undefined fragments are marked with the symbol \times .

Fragment	pure	+while	+Y0	+Y1
IA_0	PTIME	\times	\times	\times
IA_1	coNP	PSPACE	DPDA EQUIV	\times
IA_2	PSPACE	PSPACE	DPDA EQUIV	undecidable
IA_3	EXPTIME	EXPTIME	DPDA EQUIV	undecidable
$IA_i, i \geq 4$	undecidable	undecidable	undecidable	undecidable

Table 2.6: The complete complexity classification for observational equivalence in IA.

The coNP and PSPACE results are due to Murawski [Mur05].

Chapter 3

The Safe Lambda Calculus

As mentioned in the background chapter, *safety* was originally introduced as a syntactical restriction for higher-order grammars in order to study decidability of Monadic Second Order theory over infinite trees generated by this kind of device [KNU02]. The good algorithmic properties of safety in the setting of higher-order grammars motivate further investigations in the more general setting of the simply typed lambda calculus. This chapter presents an adaptation and generalization of this syntactic restriction to the lambda calculus, giving rise to what we call the “*safe lambda calculus*”.

In the first part we introduce the typing system of the safe lambda calculus. We have remarked in the background chapter, that a higher-order grammar can be just viewed as closed simply typed lambda term. This term, however, is of a particular shape due to the structure of grammar’s rules: the right-hand side of a rule is an *applicative* term (*i.e.*, containing no lambda abstraction) of ground type. An adaptation of safety to the lambda calculus setting ought to handle all possible terms, including those containing lambda-abstraction. Our definition of safety is thus adjusted accordingly.

The typing system of the safe lambda calculus is a small variation of the simply typed lambda calculus where the abstraction rule is able to abstract more than one variable at a time but with an extra constraint: the free variables in the resulting term must have order greater than the term itself. The application rule is similarly constrained. The connection with safe higher-order grammars is then made evident by restricting our calculus to pure applicative term: an applicative term of ground type is typable in the safe lambda calculus if and only if it is safe in the sense of Knapik et al.

We study how terms of this language behave with respect to the term conversions commonly studied in the lambda calculus. We adapt the notion of beta-reduction to ensure that a version of the context-reduction lemma holds (safe terms reduce to safe terms) and we show that taking the eta-long normal form preserves safety.

Next, in an attempt to quantify the impact of the safety constraint, we look at the complexity of the beta-equivalence problem—Given two safe terms, are they beta-equivalent?. The problem is known to be non-elementary for unrestricted terms [Sta79b]. We show PSPACE-hardness for the safe case by reduction from the True Quantifier Boolean Formula problem (TQBF). This PSPACE-complete problem is encodable in the order-3 fragment of the simply typed lambda calculus, but our encoding in the safe lambda calculus makes use of the entire type-hierarchy. We conjecture the problem to be elementary.

The loss of expressivity caused by safety is then characterized in terms of numeric function representable: We show that they are precisely the polynomials *without* the conditional operator. We then give a similar characterization in terms of word-functions representable.

We then consider classical typing problems in the setting of the safe lambda calculus: we show that type-checking and typability are decidable and we observe that type inhabitation is (at least) semi-decidable.

We conclude the chapter by looking at extensions of the simply typed lambda calculus. We

show how the safety restriction can be defined for languages featuring recursion and imperative feature, and we defined the safe fragments of PCF and Idealized Algol.

Related work: A first attempt to adapt the safety restriction to the lambda calculus was made by Aehlig et al. in an unpublished technical report [AdMO04]. The calculus that we present here is both simpler (the typing system is just a slight variation of the simply typed lambda calculus) and more general (no condition is imposed on types and use of Σ -constants of any order is allowed) than the one proposed by Aehlig et al.

3.1 Definition and properties

3.1.1 Safety adapted to the lambda calculus

We use sequents of the form $\Gamma \vdash_s M : A$ to represent term-in-context where Γ is a typing-context (a consistent set of typing assumptions), A is the type and M is a term (either annotated or untyped). We will introduced various subscripts s to represent terms-in-context from different typing systems. The subscript ‘st’ refers to the (Curry-style or Church-style) simply typed lambda calculus (See Convention 3.1.1.)

We fix an atomic type symbol o and for any natural number $n \in \mathbb{N}$ we use *type* notation n to refer to the type n_o defined in Sec. 2.1.5 ($0 \equiv o$ and $(k+1) \equiv k \rightarrow o$ for $k \geq 0$). A type $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$, where B is not necessarily ground, will be written (A_1, \dots, A_n, B) .

Definition 3.1.1 (The safe lambda calculus).

- (i) The **safe lambda calculus à la Curry**, denoted “safe $\Lambda_{\rightarrow}^{\text{Cu}}$ ”, is a sub-system of the simply typed lambda calculus à la Curry. It is defined as the set of judgments of the form $\Gamma \vdash_s M : A$, where M ranges over untyped term in Λ , that are derivable from the system of rules of Table 3.1.
- (ii) The **safe lambda calculus à la Church**, denoted “safe $\Lambda_{\rightarrow}^{\text{Ch}}$ ”, is the typing system obtained by adding type annotations in the λ -binders in the abstraction rule of the safe lambda calculus à la Curry (see Sec. 2.1.7). In this system M ranges over annotated term in Λ_{T} .
- (iii) The sub-systems defined by the same rules in (i) and (ii), such that all types that occur in them are homogeneous (Sec. 2.2.2), are called the **homogeneous safe lambda calculus à la Curry** and **à la Church** respectively.

(var)	$\frac{}{x : A \vdash_s x : A}$	(wk)	$\frac{\Gamma \vdash_s M : A}{\Delta \vdash_s M : A} \quad \Gamma \subset \Delta$	(δ)	$\frac{\Gamma \vdash_s M : A}{\Gamma \Vdash_{\text{app}} M : A}$
(app _{as})	$\frac{\Gamma \vdash_s M : (A_1, \dots, A_n, B) \quad \Gamma \vdash_s N_1 : A_1 \quad \dots \quad \Gamma \vdash_s N_n : A_n}{\Gamma \Vdash_{\text{app}} M N_1 \dots N_n : B}$				
(app)	$\frac{\Gamma \vdash_s M : (A_1, \dots, A_n, B) \quad \Gamma \vdash_s N_1 : A_1 \quad \dots \quad \Gamma \vdash_s N_n : A_n}{\Gamma \vdash_s M N_1 \dots N_n : B} \quad \text{ord } \Gamma \geq \text{ord } B$				
(abs)	$\frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \Vdash_{\text{app}} M : B}{\Gamma \vdash_s \lambda x_1 \dots x_n. M : (A_1, \dots, A_n, B)} \quad \text{ord } \Gamma \geq \text{ord } (A_1, \dots, A_n, B)$				

where $\text{ord } \Gamma$ denotes the set $\{\text{ord } A \mid y : A \in \Gamma\}$ and for $S \subseteq \mathbb{N}$, $u \in \mathbb{N}$, “ $S \geq u$ ” means that u is a lower-bound of S .

Table 3.1: The safe lambda calculus à la Curry.

We will consider extension of the safe lambda calculus with constants. For any set Ξ of higher-order constants, we introduce sequents of the form $\Gamma \vdash_{\mathfrak{s}}^{\Xi} M : A$ to denote the typing system obtained by adding the rule:

$$(\text{const}) \frac{}{\vdash_{\mathfrak{s}}^{\Xi} f : A} f \in \Xi .$$

For convenience, we shall omit the superscript from $\vdash_{\mathfrak{s}}^{\Xi}$ whenever the set of constants Ξ is clear from the context.

The safe lambda calculus deviates from the standard definition of the simply typed lambda calculus in a number of ways. First the application and abstraction rules can respectively perform multiple applications and abstract several variables at once. (Of course this feature alone does not alter expressivity.) Crucially, the side conditions in the application rule and abstraction rule require the variables in the typing context to have orders no smaller than that of the term being formed. Safe terms can be applied together using the rule (**app_{as}**), but the resulting term is only “almost-safe”; it can then be turned into a safe term using the abstraction rule. We do not impose any constraint on types. In particular, type-homogeneity, which was an assumption of the original definition of safe grammars [KNU02], is not required here. Another difference is that we allow the addition of Ξ -constants with arbitrary higher-order types.

Definition 3.1.2 (Safe terms).

- (i) An *untyped* term $M \in \Lambda$ is **safe** if the judgment $\Gamma \vdash_{\mathfrak{s}} M : T$ is derivable in the safe lambda calculus *à la* Curry for some context Γ and type T . Otherwise it is said to be **unsafe**.
- (ii) A *type-annotated* term $M \in \Lambda_{\mathbb{T}}$ is **safe** if the judgment $\Gamma \vdash_{\mathfrak{s}} M : T$ is derivable in the safe lambda calculus *à la* Church for some context Γ and type T . Otherwise it is said to be **unsafe**.
- (iii) An *untyped* term $M \in \Lambda$ is **universally safe** if all its valid type annotations are safe (*i.e.*, for any $M' \in \Lambda_{\mathbb{T}}$, context Γ and type A such that $\Gamma \vdash_{\text{Ch}} M' : A$ and $|M'| \equiv M$, M' is safe). It is **universally unsafe** if all its valid type annotations are unsafe.
- (iv) A term M that is typable as $\Gamma \vdash_{\text{app}} M : T$ for some Γ, T is called an **almost safe application**.
- (v) A *term-in-context* $\Gamma \vdash_{\text{st}} M : T$ of the Curry-style (resp. $\Gamma \vdash_{\text{Ch}} M : T$ of the Church-style) simply typed lambda calculus is said to be **safe** if $\Gamma \vdash_{\mathfrak{s}} M : T$ is also typable in the Curry-style (resp. Church-style) safe lambda calculus.

CONVENTION 3.1.1 To avoid cumbersome notations, we will use sequents of the form $\Gamma \vdash_{\mathfrak{s}} M : A$ to refer to judgments of both versions of the safe lambda calculus (Curry and Church). When we specify that M is an untyped term in Λ then it is understood that the judgement refers to a term-in-context typed in the Curry-style safe lambda calculus. If M ranges over annotated terms in $\Lambda_{\mathbb{T}}$ then it refers to a term-in-context typed in the Church-style safe lambda calculus. When the domain of M is not specified then it means that the argument (resp. definition, lemma or proposition) is valid in both systems.

Example 3.1.1 (Kierstead terms). Consider the annotated terms $M_1 \equiv \lambda f^2.f(\lambda x^0.f(\lambda y^0.y))$ and $M_2 \equiv \lambda f^2.f(\lambda x^0.f(\lambda y^0.x))$. M_2 is unsafe because in the subterm $f(\lambda y^0.x)$, the free variable x has order 0 which is smaller than $\text{ord}(\lambda y^0.x) = 1$. On the other hand, M_1 is safe as the following

proof tree shows:

$$\begin{array}{c}
 \text{(var)} \frac{}{f : 2 \vdash_s f : 2} \quad \text{(wk)} \frac{f : 2 \vdash_s f : 2}{f : 2, x : o \vdash_s f : 2} \quad \text{(abs)} \frac{\frac{\frac{}{y : o \vdash_s y : o} \text{(var)}}{y : o \vdash_{\text{app}} y : o} (\delta)}{y : o \vdash_s \lambda y^o. y : 1_o} \quad \text{(wk)} \frac{f : 2, x : o \vdash_s \lambda y^o. y : 1_o}{f : 2, x : o \vdash_s f(\lambda y^o. y) : o} \quad \text{(app)} \\
 \text{(var)} \frac{}{f : 2 \vdash_s f : 2} \quad \text{(abs)} \frac{\frac{f : 2 \vdash_s f(\lambda x^o. f(\lambda y^o. y)) : o}{f : 2 \vdash_s \lambda x^o. f(\lambda y^o. y) : (o, o)}}{f : 2 \vdash_s f(\lambda x^o. f(\lambda y^o. y)) : o} \quad \text{(app)} \\
 \text{(abs)} \frac{}{\vdash_s M_1 \equiv \lambda f^2. f(\lambda x^o. f(\lambda y^o. y)) : 3}
 \end{array}$$

Now consider the untyped terms underlying M_1 and M_2 : $|M_1| \equiv \lambda f. f(\lambda x. f(\lambda y. y))$ and $|M_2| \equiv \lambda f. f(\lambda x. f(\lambda y. x))$ both have for principal type $\alpha_3 \equiv ((\alpha \rightarrow \alpha) \rightarrow \alpha) \rightarrow \alpha$. Further, every typing derivation for $|M_1|$ and $|M_2|$ in the simply-typed lambda calculus assigns the same type α to the occurrences of the variables x and y . Hence $|M_1|$ is *universally safe* and $|M_2|$ is *universally unsafe*.

Example 3.1.2. The term-in-context $f : (1, 1, o) \vdash_{\text{app}} (\lambda \varphi^2 \theta^3. \varphi(\lambda x^o. x))(f(\lambda x^o. x)) \equiv M : 3$ is almost safe. Abstracting f produces the safe term-in-context $\vdash_s \lambda f^{(1,1,o)}. M : ((1, 1, o), 3)$.

The basic properties of the type lambda calculus also hold in the safe lambda calculus:

Lemma 3.1.1. (i) $\Gamma \vdash_s M : B \wedge \Gamma \subseteq \Gamma' \implies \Gamma' \vdash_s M : B$

(ii) $\Gamma \vdash_s M : B \implies FV(M) \subseteq \text{dom}(\Gamma)$

(iii) $\Gamma \vdash_s M : B \implies \Gamma_M \vdash_s M : B$ where $\Gamma_M = \{z : A \in \Gamma \mid z \in FV(M)\}$.

Proof. Trivial. □

It is easy to see that valid typing judgements of the safe lambda calculus satisfy the following simple invariant that we will later refer as the “basic property of the safe lambda calculus”:

Lemma 3.1.2 (Basic property). *Let $\Gamma \vdash_s M : B$ be a valid judgment of the Curry or Church-like safe lambda calculus. Then*

$$\forall z : A \in \Gamma : z \in FV(M) \implies \text{ord } A \geq \text{ord } B.$$

Note that the converse does not hold: take the annotated term $\lambda y^o z^o. (\lambda x^o. y)z$. Since it is closed, it trivially satisfies the condition in the conclusion of the previous lemma, but it is not safe because the variable y is not abstracted by the “ λx ” abstraction. Even for applicative term the converse does not hold: for instance the unsafe term-in-context $f : 2, g : 1, y : o \vdash_{\text{st}} f(\underline{gy}) : o$ verifies the condition of the lemma.

Subterms

Contrary to the simply typed lambda calculus, the Subterm Lemma does not hold: a safe term may contain unsafe subterms. For instance the term $\lambda f x. f x$ is universally safe however its subterm $\lambda x. f x$ is universally unsafe. There is, however, a subclass of subterms for which this result holds:

Definition 3.1.3 (Large subterms). Let M be an untyped term, the set $\widetilde{\text{sub}}(M)$ of **large subterms** of M is defined inductively by

$$\begin{aligned}
 \widetilde{\text{sub}}(x) &= \{x\} \\
 \widetilde{\text{sub}}(MN) &= \{N\} \cup \widetilde{\text{sub}}(M) \cup \widetilde{\text{sub}}(N) \\
 \widetilde{\text{sub}}(\lambda \bar{x}. M) &= \{\lambda \bar{x}. M\} \cup \widetilde{\text{sub}}(M) \text{ where } M \text{ is not an abstraction.}
 \end{aligned}$$

The set of large subterms of an annotated type is defined identically.

Lemma 3.1.3 (Subterm lemma for safe $\Lambda_{\rightarrow}^{\text{Ch}}$ and safe $\Lambda_{\rightarrow}^{\text{Cu}}$). *Let M range over Λ or $\Lambda_{\mathbb{T}}$. Then*

$$\Gamma \vdash_{\mathbb{S}} M : T \wedge M' \in \widetilde{\text{sub}}(M) \implies \Gamma' \vdash_{\mathbb{S}} M' : T' \text{ for some } \Gamma', T'.$$

Proof. The proof is a trivial induction on the structure of the term □

When a term is unsafe, we will sometimes highlight the source of its unsafety by underlining one of its large subterm as well as some free occurrence of a variable in that subterm that does not verify the condition of the previous Lemma. (We sometimes just underline the variable.) For instance the term $\lambda f^2.f(\lambda x^o.f(\lambda y^o.\underline{x}))$ is unsafe because the subterm $\lambda y^o.x$ has order greater than the order of the variable x occurring free in it.

The applicative homogeneously-typed fragment of the safe lambda calculus captures the original notion of safety due to Knapik et al. in the context of higher-order grammars (Def. 2.2.2):

Proposition 3.1.1 (Correspondence with safe grammars). *Let $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ be a grammar and let e be an applicative term generated from the symbols in $\mathcal{N} \cup \Sigma \cup \{z_1^{A_1}, \dots, z_m^{A_m}\}$. A rule $Fz_1 \dots z_m \rightarrow e$ in \mathcal{R} is safe (in the original sense of Knapik et al.) if and only if $z_1 : A_1, \dots, z_m : A_m \vdash_{\mathbb{S}}^{\Sigma \cup \mathcal{N}} e : o$ is a valid typing judgement of the homogeneous (Curry or Church-style) safe lambda calculus.*

Proof. First we observe that since e is an applicative term, the distinction between Curry and Church-style lambda calculus does not matter. We show by induction that

1. $z_1, \dots, z_m \vdash_{\text{app}} t : A$ is a valid judgment of the homogeneous safe lambda calculus containing no abstraction if and only if in the Knapik sense, all the occurrences of unsafe subterms of t are safe occurrences.

2. $z_1, \dots, z_m \vdash_{\mathbb{S}} t : A$ is a valid judgment of the homogeneous safe lambda calculus containing no abstraction if and only if in the Knapik sense, all the occurrences of unsafe subterms of t are safe occurrences, and all parameters occurring in t have order greater than $\text{ord } t$.

The constant and variable rules are trivial. For the application case: by definition, a term $t_0 \dots t_n$ is Knapik-safe iff for all $0 \leq i \leq n$, all the occurrences of unsafe subterms of t_i are safe occurrences (in the Knapik sense), and for all $1 \leq j \leq n$, the operands occurring in t_j have order greater than $\text{ord } t_j$. The (app_{as}) rule and the induction hypothesis permit us to conclude.

Now since e is an applicative term of *ground type*, the previous result gives: $z_1, \dots, z_m \vdash_{\mathbb{S}} e : o$ is a valid judgment of the homogeneous safe lambda calculus iff all the occurrences of unsafe subterms of e are safe occurrences, which is in turn equivalent to “ $Fz_1 \dots z_m \rightarrow e$ is safe” by definition of Knapik-safety for grammar rules. □

In what sense is the safe lambda calculus *safe*?

It is an elementary fact that when performing β -reduction in the lambda calculus, one must use capture-*avoiding* substitution, which is standardly implemented by renaming bound variables afresh upon each substitution. In the safe lambda calculus, however, variable capture can never happen (as the following lemma shows). Substitution can therefore be implemented simply by capture-*permitting* replacement, without any need for variable renaming.

CONVENTION 3.1.2 (Safe variable typing convention) We say that a set Γ of typing assumptions of the form $x : A$ where x is a variable and T is a simple type is **order-consistent** if all the types assigned to a given variable are of the same order:

$$x : A_1 \in \Gamma \wedge x : A_2 \in \Gamma \implies \text{ord } A_1 = \text{ord } A_2 .$$

Let $M \in \Lambda_{\mathbb{T}}$ be an annotated term. We define the set $\text{Ass}(M)$ as the set of type-assignments induced by the type annotations in M :

$$\begin{aligned} \text{Ass}(x) &= \emptyset \\ \text{Ass}(M N) &= \text{Ass}(M) \cup \text{Ass}(N) \\ \text{Ass}(\lambda x^T.M) &= \{x : T\} \cup \text{Ass}(M) . \end{aligned}$$

A Church-like term-in-context $\Gamma \vdash_{\text{Ch}} M : T$ is said to be **order-consistent** just if the set $\text{Ass}(M)$ is, and a countable set of terms M_0, M_1, \dots is **order-consistent** just if $\bigcup_{i \geq 0} \text{Ass}(M_i)$ is.

We now adopt the **safe variable typing convention**: In any definition, theorem or proof involving countably many terms, it is assumed that the set of terms involved is order-consistent.

In the following, we write $M \{N/x\}$ to denote the capture-*permitting* substitution that textually replaces all free occurrences of x in M by N without performing variable renaming (See Def. 2.1.3) and $M \{\overline{N}/\overline{x}\}$ to refer to the simultaneous version (Def. 2.1.5).

Lemma 3.1.4 (No-variable-capture lemma). *In the safe lambda-calculus à la Church, there is no variable capture when performing simultaneous capture-permitting substitution provided that we adopt the safe variable typing convention (Convention 3.1.2): if $\Gamma, \overline{x} : \overline{B} \vdash_s M : A$, $\Gamma \vdash_s N_1 : B_1, \dots, \Gamma \vdash_s N_n : B_n$, where $|\overline{x}| = n$ then*

$$M \{\overline{N}/\overline{x}\} \equiv M[\overline{N}/\overline{x}] .$$

Proof. We prove the result by structural induction on M . The variable, constant and weakening cases are trivial. Otherwise, M is of the form $\lambda \overline{y}^{\overline{C}}. M_0 \dots M_m$ where $\overline{y} = y_1 \dots y_p$, $m, p \geq 0$ and for every $0 \leq i \leq m$, M_i is safe. The simultaneous capture-permitting substitution gives:

$$M \{\overline{N}/\overline{x}\} \equiv \lambda \overline{y}^{\overline{C}}. M_0 \{\overline{N} \upharpoonright I/\overline{x} \upharpoonright I\} \dots M_m \{\overline{N} \upharpoonright I/\overline{x} \upharpoonright I\}$$

where $I = \{i \in 1..n \mid x_i \notin \overline{y}\}$ and for any list s , $s \upharpoonright I$ denotes the sublist of s obtained by keeping only elements in s whose position index in the list belongs to I .

Suppose for contradiction that a variable capture occurs in $M \{\overline{N}/\overline{x}\}$. By the induction hypothesis there is no variable capture in $M_i \{\overline{N} \upharpoonright I/\overline{x} \upharpoonright I\}$ for $0 \leq i \leq m$. This means that we are in the following situation: for some $i \in I$ and $1 \leq j \leq p$ the variable y_j occurs freely in N_i , and x_i occurs freely in M . Since $y_j \in FV(N_i)$ we must have $y_j : D \in \Gamma$ for some type D , and by the *safe variable typing convention*, we necessarily have $\text{ord } D = \text{ord } C_j$. Therefore:

$$\begin{aligned} \text{ord } D &= \geq \text{ord } B_i && \text{by Lemma 3.1.2 since } y_j \in FV(N_i), \\ &\geq \text{ord } A && \text{by Lemma 3.1.2 since } x_i \in FV(M), \\ &= 1 + \max\{\text{ord } C_k \mid 1 \leq k \leq p\} \\ &> \text{ord } C_j \\ &= \text{ord } D && \text{by the safe variable typing convention.} \end{aligned}$$

which gives us a contradiction. \square

Alternatively, one can state a slight variation of the No-variable capture Lemma that does not rely on Convention 3.1.2:

Lemma 3.1.5. *Let $\Gamma, \overline{x} : \overline{B} \vdash_s M : A$, $\Gamma \vdash_s N_1 : B_1, \dots, \Gamma \vdash_s N_n : B_n$, with $|\overline{x}| = n$, be valid judgement of the safe lambda-calculus à la Church. Then if further $\Gamma \vdash_{\text{Ch}} M \{\overline{N}/\overline{x}\} : A$ is valid Church simply typed term-in-context (not-necessarily safe) then*

$$M \{\overline{N}/\overline{x}\} \equiv M[\overline{N}/\overline{x}] .$$

Proof. The proof is the same as for the previous Lemma except that to show that $\text{ord } C_j = \text{ord } C$ we use the assumption $\Gamma \vdash_{\text{Ch}} M \{\overline{N}/\overline{x}\} : A$ instead of the safe typing convention: Since the annotated term $\lambda \overline{y}^{\overline{C}}. M_0 \{\overline{N} \upharpoonright I/\overline{x} \upharpoonright I\} \dots M_m \{\overline{N} \upharpoonright I/\overline{x} \upharpoonright I\}$ is typable in the Church-like lambda calculus, the free variables y_j in N_i must be bound by the abstraction $\lambda \overline{y}^{\overline{C}}$. Consequently its type must be C_j . Hence $D \equiv C_j$ and $\text{ord } D = \text{ord } C_j$. \square

REMARK 3.1.1 A version of the No-variable-capture Lemma also holds in safe grammars, as is implicit in (for example Lemma 3.2 of) the original paper [KNU02].

Example 3.1.3. In order to contract the β -redex in

$$f : (o, o, o), x : o \vdash_{\text{Ch}} (\lambda \varphi^{(o,o)} x^o. \varphi x)(\underline{f x}) : (o, o)$$

one should rename the bound variable x to a fresh name to prevent the capture of the free occurrence of x in the underlined subterm during substitution. Consequently, by the previous lemma, the term is not safe. And indeed the basic property of the safe lambda calculus is not verified because $\text{ord } x = 0 < 1 = \text{ord } fx$.

Note that lambda-terms that do not require variable-capture when being reduced are not necessarily safe. For instance the β -redex in $\lambda y^o z^o. (\lambda x^o. y)z$ can be contracted using capture-permitting substitution, even though the term is not safe.

Lemma 3.1.6 (Substitution Lemma). *Let $\Gamma \vdash_s N : A$. Then*

- (i) $\Gamma, x : A \vdash_s M : B \implies \Gamma \vdash_s M[N/x] : B$,
- (ii) $\Gamma, x : A \vdash_{\text{app}} M : B \implies \Gamma \vdash_{\text{app}} M[N/x] : B$.

Further if $\Gamma \vdash_s N : A$ and $\Gamma \vdash_s M : A$ are homogeneously safe then so is $\Gamma \vdash_s M[N/x] : B$, and if $\Gamma \vdash_s N : A$ and $\Gamma \vdash_s M : A$ are homogeneously almost-safe then so is $\Gamma \vdash_s M[N/x] : B$.

Proof. Let $\Gamma \vdash_s N : A$. We show (i) and (ii) simultaneously by induction on the derivation tree of $\Gamma, x : A \vdash_s M : B$ or $\Gamma, x : A \vdash_{\text{app}} M : B$. The base cases (var) and (const) are trivial. The cases (δ) and (wk) follow immediately from the induction hypothesis.

Case (abs): We have $\Gamma, x : A \vdash_s \lambda \bar{y}^{\bar{C}}. Q \equiv M : (\bar{C}, D)$. Suppose that x belongs to \bar{y} then the substitution is not pushed inside the lambda so the result holds trivially. Otherwise suppose that $\Gamma, x : A, \bar{y} : \bar{C} \vdash_{\text{app}} Q : D$. Applying the induction hypothesis (ii) on this term-in-context gives: $\Gamma, \bar{y} : \bar{C} \vdash_{\text{app}} Q[N/x] : D$ and by the rule (abs) we obtain: $\Gamma \vdash_s \lambda \bar{y}^{\bar{C}}. Q[N/x] : (\bar{C}, D)$. We can then conclude since $\lambda \bar{y}^{\bar{C}}. Q[N/x] \equiv (\lambda \bar{y}^{\bar{C}}. Q)[N/x]$ under the *safe variable naming convention* (Convention 3.1.2).

Case (app_{as}): We have $M \equiv M_0 M_1 \dots M_p$ for $p \geq 1$ and $\Gamma \vdash_s M_k : A_k$ for $1 \leq k \leq p$. By the induction hypothesis, we have $\Gamma \vdash_s M_k[N/x] : A_k$ for all k . The rules (app_{as}) permit us to conclude.

Case (app): again it is proved by applying the induction hypothesis on the premises of the rules.

Finally, term substitution preserves types so in particular it preserves type homogeneity. \square

REMARK 3.1.2 (i) This result naturally extends to simultaneous substitution: If $\Gamma \vdash_s N_k : A_k$ for $1 \leq k \leq n$ then $\Gamma, x_1 : A_1, \dots, x_n : A_n \vdash_s M : B$ implies $\Gamma \vdash_s M[N_1/x_1, \dots, N_n/x_n] : B$. and $\Gamma, x_1 : A_1, \dots, x_n : A_n \vdash_{\text{app}} M : B$ implies $\Gamma \vdash_{\text{app}} M[N_1/x_1, \dots, N_n/x_n] : B$.

- (ii) Observe that the *type* substitution lemma of the simply typed lambda calculus does not hold in the safe lambda calculus. This is because type substitution allows one to alter the order of the variables occurring in the term. For instance take $M \equiv \lambda f y. f(\lambda x. y)$. Its principal type in the lambda calculus is $A \equiv ((\alpha \rightarrow \beta) \rightarrow \gamma) \rightarrow \beta \rightarrow \gamma$ for some atomic types α, β and γ . The judgement $\vdash_{\text{st}} M : A$ is not safe (because then $\text{ord } y = \text{ord } x$), the judgment $\vdash_{\text{st}} M : A[\beta/\beta \rightarrow \beta]$ is safe, and the judgment $\vdash_{\text{st}} M : A[\beta/\beta \rightarrow \beta][\alpha/\alpha \rightarrow \alpha]$ is unsafe.

3.1.2 Safe beta reduction

It is desirable to have an appropriate notion of reduction for our calculus. The standard β -reduction rule is not adequate, however, because safety is not preserved by β -reduction as the following example shows: The safe term $\lambda f^{(o,o,o)} z^o w^o. (\lambda x^o y^o. fxy)zw$ β -reduces in one step to $\lambda f^{(o,o,o)} z^o w^o. (\lambda y^o. fzy)w$, which is unsafe since the underlined order-1 subterm contains a free occurrence of a ground variable. But if we perform one more reduction we obtain the safe term

$\lambda f^{(o,o,o)} z^o w^o . f z w$. This suggests simultaneous contraction of “consecutive” β -redexes. In order to define this notion of reduction we first introduce the corresponding notion of redex.

In the simply typed lambda calculus a redex is a term of the form $(\lambda x.M)N$. In the safe lambda calculus, a redex is a succession of several standard redexes:

Definition 3.1.4 (Safe redex). An *untyped safe redex* is an untyped almost safe application of the form $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ for some $l, n \geq 1$ such that M is an almost safe application. (Consequently $\lambda x_1 \dots x_n.M$ is safe and each N_i is safe for $1 \leq i \leq n$.) The notion of *annotated* safe redex is defined similarly.

For instance, in the case $n < l$, a safe redex has a derivation tree of the following form:

$$\begin{array}{c} \text{(abs)} \frac{\dots}{\Gamma', \bar{x} : \bar{A} \vdash_s M : (A_{n+1}, \dots, A_l, B)} \\ \text{(wk)} \frac{\Gamma' \vdash_s \lambda \bar{x}.M : (A_1, \dots, A_l, B)}{\Gamma \vdash_s (\lambda \bar{x}.M)N_1 \dots N_l : B} \quad \frac{\dots}{\Gamma \vdash_s N_1 : A_1} \quad \frac{\dots}{\Gamma \vdash_s N_l : A_l} \\ \text{(app)} \end{array}$$

where the abbreviations \bar{x} and $\bar{x} : \bar{A}$ stand for $x_1 \dots x_n$ and $x_1 : A_1, \dots, x_n : A_n$ respectively.

Example 3.1.4. The term $(\lambda f^1.((\lambda g^1 h^1.h)(\lambda z^o.z)))(\lambda z^o.z)(\lambda z^o.z)$ is a safe redex of type $o \rightarrow o$. This example shows that there exist safe redexes of the form $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ with $l > n$.

A *safe redex* is by definition an almost term, but it is not necessarily a *safe term*. For instance the term $(\lambda x^o y^o.x)z$ is a safe redex but it is only an *almost* safe term. The reason why we call such redexes “safe” is because when they occur within a safe term, it is possible to contract them without braking the safety of the whole term. Before showing this result, we first need to define how to contract safe redexes:

Definition 3.1.5 (Safe redex contraction). We use the abbreviations $\bar{x} = x_1 \dots x_n$, $\bar{N} = N_1 \dots N_l$ and $\bar{y} = y_1 \dots y_m$ for $n, l, q \geq 1$. The relation β_s (when viewed as a function) is defined on the set of safe redexes as follows:

$$\begin{aligned} \beta_s = \{ & (\lambda \bar{x}.M)N_1 \dots N_l \mapsto \lambda x_{l+1} \dots x_n.M [\bar{N}/x_1 \dots x_l] \mid n > l \} \\ & \cup \{ (\lambda \bar{x}.M)N_1 \dots N_l \mapsto M [N_1 \dots N_n/\bar{x}] N_{n+1} \dots N_l \mid n \leq l \} . \end{aligned}$$

where the notation $M [R_1 \dots R_k/z_1 \dots z_k]$ denotes the simultaneous substitution (Def. 2.1.6).

Lemma 3.1.7 (β_s preserves safety). *Suppose that $M_1 \beta_s M_2$. Then*

- (i) M_2 is almost safe;
- (ii) $\Gamma \vdash_s M_1 : A \implies \Gamma \vdash_s M_2 : A$.

Proof. Let $M_1 \beta_s M_2$ for some almost safe redex M_1 and term M_2 of type A . By definition, M_1 is of the form $(\lambda x_1 \dots x_n.M)N_1 \dots N_l$ for some safe terms N_1, \dots, N_l of type B_1, \dots, B_n ; almost safe term M of type C ; and such that $(\lambda x_1 \dots x_n.M)$ is a safe term of type (B_1, \dots, B_n, C) .

- Suppose $n > l$ then $A = (B_{l+1}, \dots, B_n, C)$. (i) By the Substitution Lemma 3.1.6(ii), the term $M [\bar{N}/x_1 \dots x_l] : C$ is an almost safe application: $\Gamma, x_{l+1} : B_{l+1}, \dots, x_n : B_n \vdash_{\text{app}} M [\bar{N}/x_1 \dots x_l] : C$. Thus by definition, $\lambda x_{l+1} \dots x_n.M [\bar{N}/x_1 \dots x_l] \equiv M_2$ is almost safe.
- (ii) Suppose that M_1 is safe. W.l.o.g we can assume that the last rule used to form M_1 is (app) (and not the weakening rule (wk)), thus we have $\text{dom } \Gamma = FV(M_1)$, and Lemma 3.1.2 gives us $\text{ord } A \leq \text{ord } \Gamma$. This allows us to use the rule (abs) to form the safe term $\Gamma \vdash_s \lambda x_{l+1} \dots x_n.M [\bar{N}/x_1 \dots x_l] \equiv M_2 : A$.
- Suppose $n \leq l$. (i) Again by the Substitution Lemma we have that $M [N_1 \dots N_n/\bar{x}]$ is an almost safe application: $\Gamma \vdash_{\text{app}} M [N_1 \dots N_n/\bar{x}] : C$. If $n = l$ then the proof is finished; otherwise ($n < l$) we further apply the rule (app_{as}) $l - n$ times which gives us the almost safe application $\Gamma \vdash_{\text{app}} M_2 : A$.
- (ii) Suppose that M_1 is safe. If $n = l$ then $M_2 \equiv M [N_1 \dots N_n/\bar{x}]$ is safe by the Substitution Lemma; If $n < l$ then we obtain the judgement $\Gamma \vdash_s M_2 : A$ by applying the rule (app_{as}) $l - n - 1$ times on $\Gamma \vdash_s M [N_1 \dots N_n/\bar{x}] : C$ followed by one application of (app). \square

We can now define a notion of reduction for safe terms:

Definition 3.1.6 (Safe beta-reduction). The *safe β -reduction*, written \rightarrow_{β_s} , is the compatible closure of the relation β_s with respect to the formation rules of the safe lambda calculus (*i.e.*, it is the smallest relation such that if $M_1 \beta_s M_2$ and $C[M]$ is a safe term for some context $C[-]$ formed with the rules of the simply typed lambda calculus then $C[M_1] \rightarrow_{\beta_s} C[M_2]$). The relation $=_{\beta_s}$ is defined as the reflexive, symmetric, transitive closure of \rightarrow_{β_s} .

Lemma 3.1.8. *The safe reduction relation \rightarrow_{β_s} :*

- (i) *is a subset of the transitive closure of \rightarrow_{β} ($\rightarrow_{\beta_s} \subset \rightarrow_{\beta}$);*
- (ii) *is strongly normalizing;*
- (iii) *has the unique normal form property;*
- (iv) *has the Church-Rosser property.*

Proof. (i) Immediate from the definition: safe β -reduction is just a multi-step β -reduction. (ii) This is because $\rightarrow_{\beta_s} \subset \rightarrow_{\beta}$ and, \rightarrow_{β} is strongly normalizing in the simply typed lambda-calculus. (iii) It is easy to see that if a safe term has a beta-redex if and only if it has a safe beta-redex (because a beta-redex can always be “widen” into consecutive beta-redex of the shape of those in Def. 3.1.5. Therefore the set of β_s -normal forms is equal to the set of β -normal forms. The unicity of β -normal form then implies the unicity of β_s -normal form. (iv) is a consequence of (i) and (ii). \square

Since \rightarrow_{β_s} is by definition the compatible closure of β_s by the formation rules of the safe lambda calculus, Lemma 3.1.7 implies

Lemma 3.1.9 (Subject Reduction). *Let $M_1 \rightarrow_{\beta_s} M_2$. Then*

- (i) $\Gamma \vdash_s M_1 : B \implies \Gamma \vdash_s M_2 : B$,
- (ii) $\Gamma \Vdash_{\text{app}} M_1 : B \implies \Gamma \Vdash_{\text{app}} M_2 : B$.

Proof. Suppose that $M_1 \rightarrow_{\beta_s} M_2$. Then we have $M_1 \equiv C[R_1]$ and $M_2 \equiv C[N_2]$ for some context $C[-]$ and safe redex N_1 with $N_1 \beta_s N_2$.

(i) If the safe redex N_1 is a safe term $\Gamma' \vdash_s N_1 : A$ then by Lemma 3.1.7(ii) we have $\Gamma' \vdash_s N_2 : A$. We can therefore deduce $\Gamma \vdash_s C[N_2] \equiv M_2 : B$ by replacing the derivation subtree of $\Gamma' \vdash_s N_1 : A$ by the derivation tree of $\Gamma' \vdash_s N_2 : A$ in the derivation tree of $\Gamma \vdash_s C[N_1] : B$.

Otherwise N_1 is an almost safe application that is not safe and therefore N_1 is a strict subterm of M_1 . In the derivation tree of a safe term, an almost safe application that is not safe can only occur as a premise of the abstraction rule. Thus the context $C[-]$ must be of the form $C'[\lambda \bar{y}. -]$ for some context $C'[-]$ and such that $\lambda \bar{y}. N_1$ is a safe term: $\Gamma'' \vdash_s \lambda \bar{y}. N_1 : C$ for some Γ'', C . Applying the abstraction rule on N_2 gives $\Gamma'' \vdash_s \lambda \bar{y}. N_2 : C$. Hence as in the previous case we can deduce $\Gamma \vdash_s C[N_2] \equiv C'[\lambda \bar{y}. N_2] \equiv M_2 : B$ by substituting the derivation tree of $\Gamma'' \vdash_s \lambda \bar{y}. N_2 : C$ for the derivation tree $\Gamma'' \vdash_s \lambda \bar{y}. N_1 : C$ in the derivation tree of $\Gamma \vdash_s M_1 : B$.

(ii) If N_1 is a safe term then we conclude as in (i). Otherwise, N_1 is an almost safe application: if $C[-] \equiv -$ then we can conclude immediately by Lemma 3.1.7(i); otherwise N_1 necessarily occurs as a subterm of a safe subterm of M_1 so we can conclude as in (i). \square

REMARK 3.1.3 \rightarrow_{β_s} *does not* preserve “unsafety”: Take any safe annotated-term S and unsafe annotated-term U of the same type τ , then the term $(\lambda x^\tau y^\tau. y) U S : \tau$ is unsafe but it β_s -reduces to S which is safe.

3.1.3 Eta-long normal form

We now restrict our attention to the Church-style (safe) lambda calculus. Since terms are annotated, their type as well as the types of their subterms are uniquely determined. The η -expansion of $M : A \rightarrow B$ is defined as the annotated term $\lambda x^A.Mx : A \rightarrow B$ where $x : A$ is a fresh variable. The η -long-expansion of a term $M : (A_1, \dots, A_n, o)$ is defined as $\lambda \varphi_1^{A_1} \dots \varphi_l^{A_l}.M\varphi_1 \dots \varphi_l$ where each φ_i is a fresh variable. The η -long normal form (or just η -long form) of an annotated term (also referred in the literature as *long reduced form*, *η -normal form* or *extensional form* [JP76, Hue75, Hue76]) is obtained by hereditarily η -expanding the body of every lambda abstraction as well as every subterm occurring in an *operand position* (i.e., occurring as the second argument of some occurrence of the binary application operator). Formally,

Definition 3.1.7. The η -long form, written $\llbracket M \rrbracket$ or sometimes $\eta_{\text{nf}}(t)$, of an annotated term M of type (A_1, \dots, A_n, o) with $n \geq 0$ is defined by cases according to the syntactic shape of M (A simply typed term is either an abstraction or it can be written uniquely as $s_0 s_1 \dots s_m$ where $m \geq 0$ and s_0 is a variable, a Σ -constant or an abstraction.):

$$\begin{aligned} \llbracket \lambda x^\tau.N \rrbracket &\equiv \lambda x^\tau.\llbracket N \rrbracket \\ \llbracket \alpha N_1 \dots N_m \rrbracket &\equiv \lambda \bar{\varphi}^{\bar{A}}.\alpha \llbracket N_1 \rrbracket \dots \llbracket N_m \rrbracket \llbracket \varphi_1 \rrbracket \dots \llbracket \varphi_n \rrbracket \\ \llbracket (\lambda x^\tau.N)N_1 \dots N_p \rrbracket &\equiv \lambda \bar{\varphi}^{\bar{A}}.(\lambda x^\tau.\llbracket N \rrbracket) \llbracket N_1 \rrbracket \dots \llbracket N_p \rrbracket \llbracket \varphi_1 \rrbracket \dots \llbracket \varphi_n \rrbracket \end{aligned}$$

where $m \geq 0$, $p \geq 1$, x is a variable, $\bar{\varphi} = \varphi_1 \dots \varphi_n$ and each $\varphi_i : A_i$ is a fresh variable, and α is either a variable or a constant.

REMARK 3.1.4 The η -long normal form is defined for any simply typed lambda term, whether β -normal or not. Furthermore, the transformation does not introduce any new redex therefore the η -long normal form of a β -normal term remains β -normal.

Definition 3.1.8. We say that a safe annotated term is **long-safe** just if it is typable in the Church-like safe lambda calculus without using the rule (**app_{as}**) from Def. 3.1.1. Equivalently, it is long-safe just if the judgment $\Gamma \vdash_1 M : T$ for some Γ, T can be derived from the system of rules of Table 3.2.

$$\begin{aligned} (\text{var}_1) \quad & \frac{}{\Gamma \vdash_1 x : A} \quad x : A \in \Gamma \quad (\text{wk}_1) \quad \frac{\Gamma \vdash_1 M : A}{\Delta \vdash_1 M : A} \quad \Gamma \subset \Delta \\ (\text{app}_1) \quad & \frac{\Gamma \vdash_1 M : (A_1, \dots, A_n, B) \quad \Gamma \vdash_1 N_1 : A_1 \quad \dots \quad \Gamma \vdash_1 N_n : A_n}{\Gamma \vdash_1 MN_1 \dots N_n : B} \quad \text{ord } \Gamma \geq \text{ord } B \\ (\text{abs}_1) \quad & \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \vdash_1 M : B}{\Gamma \vdash_1 \lambda x_1^{A_1} \dots \lambda x_n^{A_n}.M : (A_1, \dots, A_n, B)} \quad \text{ord } \Gamma \geq \text{ord } (A_1, \dots, A_n, B) \end{aligned}$$

Table 3.2: Typing rules for long-safe terms-in-contexts.

The terminology “long-safe” is deliberately suggestive of a forthcoming lemma. Note that long-safe terms are not necessarily in η -long normal form. By definition, if an annotated term is long-safe then it is safe:

Lemma 3.1.10. $\Gamma \vdash_1 M : T \implies \Gamma \vdash_s M : T$.

In general, long-safety is not preserved by η -expansion: for instance we have $\vdash_1 \lambda y^o z^o.y : (o, o, o)$ but $\nvdash_1 \lambda x^o.(\lambda y^o z^o.y)x : (o, o, o)$. On the other hand, η -reduction (of one variable) preserves long-safety:

Lemma 3.1.11. $\Gamma \vdash_1 \lambda \varphi^\tau.M\varphi : A \wedge \varphi \notin FV(M) \implies \Gamma \vdash_1 M : A$.

Proof. Suppose $\Gamma \vdash_1 \lambda\varphi^\tau.M\varphi : A$. If s is an abstraction then by construction the annotated-term s is necessarily safe. If $M \equiv N_0 \dots N_p$ with $p \geq 1$ then again, since $\lambda\varphi^\tau.N_0 \dots N_p\varphi$ is safe, each of the N_i is safe for $0 \leq i \leq p$ and for any $z \in FV(\lambda\varphi^\tau.M\varphi)$, $\text{ord } z \geq \text{ord } \lambda\varphi^\tau.M\varphi = \text{ord } s$. Since φ does not occur free in M we have $FV(M) = FV(\lambda\varphi^\tau.M\varphi)$, thus we can use the application rule to form $\Gamma_M \vdash_1 N_0 \dots N_p : A$ where Γ_M is the subset of Γ verifying $\text{dom}(\Gamma) = FV(M)$. The weakening rules permits us to conclude $\Gamma \vdash_1 M : A$. \square

Lemma 3.1.12 (Long-safety is preserved by η -long expansion). $\Gamma \vdash_1 M : A \implies \Gamma \vdash_1 [M] : A$.

Proof. First we observe that for any variable or constant $x : A$ we have $x : A \vdash_1 [x] : A$. We show this by induction on $\text{ord } x$. It is verified for any ground type variable x since $x = [x]$. Step case: $x : A$ with $A = (A_1, \dots, A_n, o)$ and $n > 0$. Let $\varphi_i : A_i$ be fresh variables for $1 \leq i \leq n$. Since $\text{ord } A_i < \text{ord } x$ the induction hypothesis gives $\varphi_i : A_i \vdash_1 [\varphi_i] : A_i$. Using (wk_l) we obtain $x : A, \overline{\varphi} : \overline{A} \vdash_1 [\varphi_i] : A_i$. The application rule gives $x : A, \overline{\varphi} : \overline{A} \vdash_1 x[\varphi_1] \dots [\varphi_n] : o$ and the abstraction rule gives $x : A \vdash_1 \lambda\overline{\varphi}.x[\varphi_1] \dots [\varphi_n] \equiv [x] : A$.

We now prove the lemma by induction on s . The base case is covered by the previous observation. Step case:

- $M \equiv xN_1 \dots N_m$ with $x : (B_1, \dots, B_m, A)$, $A = (A_1, \dots, A_n, o)$ for some $m \geq 0$, $n > 0$ and $N_i : B_i$ for $1 \leq i \leq m$. Let $\varphi_i : A_i$ be fresh variables for $1 \leq i \leq n$. By the previous observation we have $\varphi_i : A_i \vdash_1 [\varphi_i] : A_i$, the weakening rule then gives us $\Gamma, \overline{\varphi} : \overline{A} \vdash_1 [\varphi_i] : A_i$. Since the judgement $\Gamma \vdash_1 xN_1 \dots N_m : A$ is formed using the (app_l) rule, each N_j must be long-safe for $1 \leq j \leq m$, thus by the induction hypothesis we have $\Gamma \vdash_1 [N_j] : B_j$ and by weakening we get $\Gamma, \overline{\varphi} : \overline{A} \vdash_1 [N_j] : B_j$. The (app_l) rule then gives $\Gamma, \overline{\varphi} : \overline{A} \vdash_1 x[N_1] \dots [N_m][\varphi_1] \dots [\varphi_n] : o$. Finally the (abs_l) rule gives $\Gamma \vdash_1 \lambda\overline{\varphi}.x[N_1] \dots [N_m][\varphi_1] \dots [\varphi_n] \equiv [M] : A$, the side-condition of (abs_l) being verified since $\text{ord } [M] = \text{ord } M$.
- $M \equiv N_0 \dots N_m$ where $m \geq 1$ and N_0 is an abstraction. The eta-long normal form of M is $[M] \equiv \lambda\overline{\varphi}.[N_0] \dots [N_m][\varphi_1] \dots [\varphi_n]$ for some fresh variables $\varphi_1, \dots, \varphi_n$. Again, using the induction hypothesis we can easily derive $[s] : A$.
- $M \equiv \lambda\overline{\eta}^{\overline{B}}.N$ where $A = (\overline{B}, C)$ and N is not an abstraction. The induction hypothesis gives $\Gamma, \overline{\eta} : \overline{B} \vdash_1 [N] : C$ and using (abs_l) we get $\Gamma \vdash_1 \lambda\overline{\eta}.[N] \equiv [M] : A$. \square

REMARK 3.1.5

1. The converse of this lemma does not hold in general: Performing η -reduction over a large abstraction does not in general preserve long-safety. This does not contradict Lemma 3.1.11 which states that safety is preserved when performing η -reduction on an abstraction of a *single* variable. The simplest counter-example is the term $f^{(o,o,o)} \vdash_{\text{st}} \lambda x^o.f\overline{x}$ which is not long-safe and whose eta-long normal form $f^{(o,o,o)} \vdash_1 \lambda x^o y^o.fxy$ is long-safe. Even for closed terms the converse does not hold: $\lambda f^{(o,o,o)} g^{((o,o,o),o)}.g(\lambda x^o y^o.f\overline{x})$ is not long-safe but its eta-normal form $\lambda f^{(o,o,o)} g^{((o,o,o),o)}.g(\lambda x^o y^o.fxy)$ is long-safe. In fact even the closed $\beta\eta$ -normal term $\lambda f^{(o,(o,o),o,o)} g^{((o,o),o,o,o),o)}.g(\lambda y^{(o,o)} x^o.f\overline{xy})$ which is not long-safe has a long-safe η -long normal form!
2. In an eta-long normal term, applications occurring in it can always be chosen large enough so that the side-condition of the rule (app_l) is verified. Hence if a term remains not long-safe after η -long expansion, then it must be due to some abstraction in the term that cannot be typed.

Lemma 3.1.13. An annotated term $M \in \lambda_{\text{T}}$ is safe if and only if its η -long normal form is long-safe; formally:

$$\Gamma \vdash_s M : T \iff \Gamma \vdash_1 [M] : T.$$

Proof. (Only if) Let $\Gamma \vdash_s M : (A_1, \dots, A_l, o)$. We show the result by induction on the structure of M . The base cases and weakening case are trivial. Abstraction: M has the form $\lambda\overline{\eta}.M_0 \dots M_p$

for some safe terms M_k , $0 \leq k \leq p$, $p \geq 0$. By the subject reduction lemma we have $\Gamma_M \vdash_s M : (A_1, \dots, A_l, o)$ where Γ_M is the subset of Γ containing only typing for free variables in M . The η -long expansion of M is $\lambda \bar{y} x_1 \dots x_l. [M][x_1] \dots [x_l]$ for some variables $x_1 : A_1, \dots, x_l : A_l$ fresh in M . Let k range in $\{1..l\}$. By Lemma 3.1.12 and 3.1.10, each $[x_k]$ is safe, and by the I.H. $[M]$ is also safe. Therefore by (app_{as}), so is $[M][x_1] \dots [x_l]$. By Lemma 3.1.2, all the free variables of M have order greater than $\text{ord}(A_1, \dots, A_l, o)$, hence we can use the abstraction rule to form the judgment $\Gamma_M \vdash_s \lambda \bar{y} x_1 \dots x_l. [M][x_1] \dots [x_l] : (A_1, \dots, A_l, o)$ and the weakening rule permits us to conclude. The application case is treated identically.

(If) We proceed by induction on the structure of the Church term-in-context $\Gamma \vdash_{\text{Ch}} M : T$: The variable, constant and weakening cases are trivial. Suppose that M is an application of the form $x N_1 \dots N_m : A$ for $m \geq 1$. Its η -long normal form is of the form $x[N_1] \dots [N_m][\varphi_1] \dots [\varphi_m] : o$ for some fresh variables $\varphi_1, \dots, \varphi_m$. By assumption this term is long-safe term therefore we have $\text{ord } A \leq \text{ord } \Gamma$ and for $1 \leq i \leq m$, $[N_i]$ is also long-safe. By the induction hypothesis this implies that the N_i are all safe. We can then form the judgment $\Gamma \vdash_s x N_1 \dots N_m : A$ using the rules (var) and (app) (this is allowed since we have $\text{ord } A \leq \text{ord } \Gamma$). The case $M \equiv (\lambda x. N) N_1 \dots N_m$ for $m \geq 1$ is treated identically.

Suppose that $M \equiv \lambda \bar{x} \bar{B}. N : A$. By assumption, its η -long n.f. $\lambda \bar{x} \bar{B} \bar{\varphi} \bar{C}. [N][\varphi_1] \dots [\varphi_m] : A$ (for some fresh variables $\bar{\varphi} = \varphi_1 \dots \varphi_m$) is long-safe. Thus we have $\text{ord } A \leq \text{ord } \Gamma$. Furthermore the long-safe subterm $[M][\varphi_1] \dots [\varphi_m]$ is precisely the eta-long form of $M \varphi_1 \dots \varphi_m : o$ therefore by the induction hypothesis we have that $M \varphi_1 \dots \varphi_m : o$ is safe. Since the φ_i 's are all safe (by rule (var)), we can “peal-off” m applications of the rule (app_{as}) (or (app)) from the sequent $\Gamma, \bar{x} : \bar{B}, \bar{\varphi} : \bar{C} \vdash_s s \varphi_1 \dots \varphi_m : o$ which gives us the sequent $\Gamma, \bar{x} : \bar{B}, \bar{\varphi} : \bar{C} \vdash_{\text{app}} M : A$. Since the variables $\bar{\varphi}$ are fresh for M , we can further peal-off one application of the weakening rule to obtain the judgment $\Gamma, \bar{x} : \bar{B} \vdash_s M : A$. Finally we obtain $\Gamma \vdash_s \lambda \bar{x} \bar{B}. M : A$ using the rule (abs) (which is permitted since we have $\text{ord } A \leq \text{ord } \Gamma$). \square

Proposition 3.1.2. *An annotated term $M \in \Lambda_{\top}$ is safe if and only if its η -long normal form is safe; formally:*

$$\Gamma \vdash_s M : B \iff \Gamma \vdash_s [M] : B .$$

Proof.

$$\begin{array}{ll} \text{(If):} & \Gamma \vdash_s [M] : T \implies \Gamma \vdash_1 [M] : T \quad \text{By Lemma 3.1.13 (only if),} \\ & \implies \Gamma \vdash_s [M] : T \quad \text{By Lemma 3.1.13 (if).} \end{array}$$

$$\begin{array}{ll} \text{(Only if):} & \Gamma \vdash_s M : T \implies \Gamma \vdash_1 [M] : T \quad \text{By Lemma 3.1.13 (only if),} \\ & \implies \Gamma \vdash_s [M] : T \quad \text{By Lemma 3.1.10} \quad \square \end{array}$$

3.1.4 Almost safety

We now give an alternative presentation of the safe lambda calculus. Consider the Curry-style system of rules of Table 3.3. The Church-style version of this system is obtained by annotating the λ -binder in the abstraction rule.

It is easy to see that these (Curry-style and Church-style) systems of rules are equivalent to the ones from Def. 3.1.1 in the sense that they generate the same set of judgments of the form $\Gamma \vdash_s M : T$. The above systems, however, have the advantage of decomposing the application and abstraction rules into atomic steps where only one variable is abstracted at a time and only two terms are applied together at a time.

Definition 3.1.9. Terms typed with the entailment operator \Vdash are called *almost safe* terms. Terms typed with the entailment operator \Vdash_{app} are called *almost safe applications*.

The intuition behind these rules is that almost safe terms represent terms that are not safe but which can become safe if sufficiently many safe terms are applied to them or if sufficiently many variables are abstracted. The rule (app_{as}) is used to form applications in which each applied term is safe:

$$\begin{array}{c}
(\text{var}_{\text{as}}) \frac{}{\Gamma \vdash_{\text{app}} x : A} \quad x : A \in \Gamma \quad (\text{wk}_{\text{as}}) \frac{\Gamma \vdash_{\text{app}} M : A}{\Delta \vdash_{\text{app}} M : A} \quad \Gamma \subset \Delta \quad (\text{wk}) \frac{\Gamma \vdash_s M : A}{\Delta \vdash_s M : A} \quad \Gamma \subset \Delta \\
\\
(\text{app}_{\text{as}}) \frac{\Gamma \vdash_{\text{app}} M : A \rightarrow B \quad \Gamma \vdash_s N : A}{\Gamma \vdash_{\text{app}} M N : B} \quad (\text{abs}_{\text{as}}) \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B} \\
\\
(\delta) \frac{\Gamma \vdash_s M : A}{\Gamma \vdash_{\text{app}} M : A} \quad (\delta') \frac{\Gamma \vdash_{\text{app}} M : A}{\Gamma \vdash M : A} \quad (\rho) \frac{\Gamma \vdash M : A}{\Gamma \vdash_s M : A} \text{ ord } \Gamma \geq \text{ord } A .
\end{array}$$

Table 3.3: Alternative Curry-style definition of the safe lambda calculus.

Lemma 3.1.14.

1. If $\Gamma \vdash_{\text{app}} M : T$ then $M \equiv N_0 \dots N_m$ for some $m \geq 0$ where N_i is safe for every $0 \leq i \leq m$;
2. If $\Gamma \vdash M : T$ then $M \equiv \lambda x_1 \dots x_n. N_0 \dots N_m$ for some $n, m \geq 0$ where N_i is safe for every $0 \leq i \leq m$.

This results follows immediately from the definition of the rules.

The rule (abs_{as}) is nothing less than the standard abstraction rule of the lambda calculus. As soon as the context and the type of the term being formed respect the safety condition (*i.e.*, all the context variables have order greater than the order of the type), the term can be marked as safe. This is done using the rule (ρ) . Together with the rule (δ') this implies that the closure of an almost safe term is always safe:

Lemma 3.1.15. $\Gamma \vdash_{\text{app}} M : T \wedge \text{dom}(\Gamma) = \text{FV}(M) \implies \vdash_s \text{closure}(M) : T$.

The two weakening rules (wk) and (wk_{as}) permit one to extend the context of a safe term or an almost safe application. We could have added a third rule to allow weakening for almost safe terms $\Gamma \vdash M : T$ as well. This is however not necessary because this kind of weakening can always be eliminated. (In particular if the term is an abstraction then we can instead apply the rule (wk_{as}) just before the abstraction rule).

An annotated term is almost safe if and only if its eta-long normal form is safe:

Lemma 3.1.16. Let $M \in \Lambda_{\mathbb{T}}$. Then $\Gamma \vdash M : T$ if and only if $\Gamma \vdash \eta_{\text{nf}}(M) : T$.

Proof. Only if: Let $\Gamma \vdash M : T$ be an almost safe term. We proceed by induction on M . Suppose that the last rule used is (δ') . Then by Lemma 3.1.14 M is an application $N_0 N_1 \dots N_k : (A_1, \dots, A_n)$ with $k \geq 0$. Let φ_i for $i \in \{1..n\}$ be fresh variables, using the rules (var_{as}) , (wk_{as}) , (app_{as}) and (abs_{as}) we can build the almost safe term $\Gamma \vdash \lambda \varphi_1^{A_1} \dots \varphi_n^{A_n}. N_0 N_1 \dots N_k \varphi_1 \dots \varphi_n : T$.

If the last rule used is (δ) then M is safe therefore by Proposition 3.1.2, its eta-long normal form is safe and therefore by (δ) it is also almost safe.

If the last rule used is (abs_{as}) then by the induction hypothesis the eta-long nf of the premise is almost safe so we can conclude using (abs_{as}) .

If: It is again a proof by structural induction on the eta-long normal form. The basic idea is that the rule (abs_{as}) (resp. $(\text{app}_{\text{app}})$) allows us to “peel-off” the lambda-abstraction (resp. extra operands) introduced during the eta-expansion. \square

The two preceding lemmas show that the closure of the eta-long normal form of an almost safe term is safe. This explains the expression “almost safe”: an almost safe is semantically safe in the sense that it is (extensionally) equivalent to a safe term; on the other hand it is syntactically unsafe since it cannot appear as an operand of an application inside a larger safe term.

Lemma 3.1.17 (Safe beta reduction preserves almost safety). Let $M \rightarrow_{\beta_s} M'$. Then

$$\Gamma \vdash M : A \implies \Gamma \vdash M' : A .$$

Proof. Suppose that $M \rightarrow_{\beta_s} M'$ and $\Gamma \vdash M : A$. By Lemma 3.1.14, $M \equiv \lambda x_1 \dots x_n. N_0 \dots N_m$ for some $n, m \geq 0$ where N_i is safe for every $0 \leq i \leq m$. There are two cases: If the reduced redex occurs in some N_i for $0 \leq i \leq m$ then we have $N \equiv \lambda x_1 \dots x_n. N_0 \dots N'_i \dots N_m$ where $N_i \rightarrow_{\beta_s} N'_i$ for some N'_i . Since safety is preserved by safe reduction (Lemma 3.1.9), N'_i is safe. Hence we can conclude using the application and abstraction rule. The second case is when the redex is $N_1 \dots N_q$ for some $1 \leq q \leq m$. This means that N_0 is of the form $\lambda y_1 \dots y_q. P$ for some safe term P , and $M' \equiv P[N_1/y_1 \dots N_q/y_q]N_{q+1} \dots N_m$. The Substitution Lemma 3.1.6 and the application and abstraction rule permit us to conclude. \square

3.1.5 Safety with respect to other type-ranking functions

We call **type-ranking function** any function $\text{rank} : \mathbb{T} \longrightarrow (L, \leq)$ mapping the set \mathbb{T} of simple types over a set of atomic types \mathbb{A} to some preorder (L, \leq) .

Example 3.1.5. The followings are examples of type-ranking functions $\mathbb{T} \longrightarrow (\mathbb{N}, \leq)$:

- The type-order defined by $\text{ord}(\alpha) = 0$ for $\alpha \in \mathbb{A}$, and $\text{ord}(A \rightarrow B) = \max(\text{ord}(A)+1, \text{ord}(B))$;
- The height of a type defined by $\text{height}(A \rightarrow B) = 1 + \max(\text{height}(A), \text{height}(B))$ and $\text{height}(\alpha) = 0$ for $\alpha \in \mathbb{A}$;
- The type-arity function defined by $\text{arity}(A_1, \dots, A_n, \alpha) = n$;
- The size of a type defined by $\text{size}(\alpha) = 0$ for $\alpha \in \mathbb{A}$ and $\text{size}(A \rightarrow B) = \text{size}(A) + \text{size}(B)$.

The pairing of two type-ranking functions is also a type-ranking function. For instance $\langle \text{ord}, \text{arity} \rangle : \mathbb{T} \longrightarrow (\mathbb{N} \times \mathbb{N}, \leq)$ is a type-ranking function where \leq denotes the lexicographic ordering.

We have defined the safe lambda calculus as a restriction on the simply typed lambda calculus obtained by restricting the occurrences of variables according to their *order*. *Would it make sense to define a version of the safe lambda calculus where the constraint uses a different type-ranking function?*

In the safe lambda calculus, the application and abstraction rules permit us to perform multiple abstraction or application at a time. For the abstraction rule, the idea is that the side-condition might not be verified after one abstraction but it may become after consecutive abstractions, and similarly for the application rule. So the design of the typing rules implicitly assume that abstracting variables increases the order of the term's type, and similarly performing application decreases its order:

$$\text{rank}(A \rightarrow B) \geq \text{rank}(B) . \quad (3.1)$$

On the other hand, in order to prove the No-variable-capture Lemma we need the following property:

$$\text{rank}(A \rightarrow B) > \text{rank}(A) . \quad (3.2)$$

The minimal function verifying the two previous equations is precisely the function ord : any function $\text{rank} : \mathbb{T} \longrightarrow (L, \leq)$ verifying (3.1) and (3.2) is greater than ord by pointwise ordering.

Hence the typing-system defining the safe lambda calculus is only of interest if the ranking function used is the type-order function ord .

3.1.6 Homogeneous safe lambda calculus

The safe lambda calculus that we have studied up to now does not make any assumption on types. In its original form however (in the setting of higher-order grammars) the safety restriction makes a further assumption on types called *homogeneity*. We recall from Section 3.1.6 that a type (A_1, \dots, A_n, o) is said to be homogeneous whenever $\text{ord } A_1 \geq \text{ord } A_2 \geq \dots \geq \text{ord } A_n$ and each of

the A_i is homogeneous. As defined in Sec. 3.1.1, the *homogeneous safe lambda calculus* denotes the restriction of the safe lambda calculus where types occurring in the derivation trees are all homogeneous. We now give a presentation of this calculus by means of a proper system of rules in which type homogeneity is implicitly enforced by the typing rules themselves.

We call **stratified context** any context of the form $x_{11} : A_{11}, \dots, x_{1r} : A_{1r}, x_{21} : A_{21}, \dots$ such that variables are listed in decreasing order and such that for any k, l and $i > j$, $\text{ord } x_{ik} > \text{ord } x_{jl}$. In other words, the context is stratified into lists of variables of the same orders, and the stratifications are arranged in strict decreasing order. Such stratified context will be abbreviated as

$$\overline{x_1} : \overline{A_1} \mid \dots \mid \overline{x_n} : \overline{A_n}$$

For any unstratified context Γ , we write $\text{strat}(\Gamma)$ to denote any possible valid stratification of Γ .

Definition 3.1.10. We define typing judgements of the form: $\overline{x_1} : \overline{A_1} \mid \dots \mid \overline{x_n} : \overline{A_n} \vdash_h M : B$ by induction over the following rules:

$$\begin{aligned}
& (\text{h-const}) \frac{}{\vdash_h f : A} \quad f : A \in \Sigma \quad (\text{h-var}) \frac{}{\overline{x_1} : \overline{A_1} \mid \dots \mid \overline{x_n} : \overline{A_n} \vdash_h x_{ij} : A_{ij}} \quad (\delta) \frac{\Gamma \vdash_h M : A}{\Gamma \vdash_{\text{h.app}} M : A} \\
& (\text{h-wk}) \frac{\Gamma \vdash_h M : B \quad \Gamma \subset \Delta}{\Delta \vdash_h M : B} \quad (\text{perm}) \frac{\Gamma \vdash_h M : B \quad \sigma(\Gamma) \text{ homogeneous}}{\sigma(\Gamma) \vdash_h M : B} \\
& (\text{h-app}_{\text{as}}) \frac{\Gamma \vdash_h s : (A_1, \dots, A_n, B) \quad \Gamma \vdash_h t_1 : A_1 \quad \dots \quad \Gamma \vdash_h t_n : A_n}{\Gamma \vdash_{\text{h.app}} s \, t_1 \dots t_n : B} \\
& (\text{h-app}_{\text{strat}}) \frac{\Gamma \vdash_h N_0 : (B_{11}, \dots, B_{1l} \mid \overline{B_2} \mid \dots \mid \overline{B_m} \mid o) \quad \Gamma \vdash_h N_1 : B_{11} \quad \dots \quad \Gamma \vdash_h N_l : B_{1l}}{\Gamma \vdash_h N_0 N_1 \dots N_l : (\overline{B_2} \mid \dots \mid \overline{B_m} \mid o)} \\
& (\text{h-app}_{\text{partial}}) \frac{\Gamma \vdash_h M : (B_{11}, \dots, B_{1l} \mid \overline{B_2} \mid \dots \mid \overline{B_m} \mid o) \quad \Gamma \vdash_h N : B_{11}}{\Gamma \vdash_h MN : (B_{12}, \dots, B_{1l} \mid \overline{B_2} \mid \dots \mid \overline{B_m} \mid o)} \quad \text{ord } \Gamma > \text{ord } B_{11} \\
& (\text{h-abs}) \frac{\overline{x_1} : \overline{A_1} \mid \dots \mid \overline{x_{p+1}} : \overline{A_{p+1}} \mid \dots \mid \overline{x_n} : \overline{A_n} \vdash_{\text{h.app}} M : B}{\overline{x_1} : \overline{A_1} \mid \dots \mid \overline{x_p} : \overline{A_p} \vdash_h \lambda \overline{x_{p+1}} \dots \overline{x_n}. M : (\overline{A_{p+1}} \mid \dots \mid \overline{A_n} \mid B)} \quad \text{ord } \overline{A_n} \geq \text{ord } B - 1
\end{aligned}$$

where Δ is an homogeneously-typed alphabet, Σ is a set of homogeneously-typed constants, and σ denotes a variable list permutation.

The main changes compared to the rules of the non-homogeneous safe lambda calculus are:

- (i) The contexts are stratified;
- (ii) All the types appearing in the rule are homogeneous;
- (iii) The rule **(h-app_{as})** is the counterpart of rule **(app_{as})** in the safe lambda calculus: it says that you can form an *homogeneous almost safe term* by applying several safe terms together;
- (iv) The original application rule **(app)** is split into two rules: (a) **(h-app_{strat})** is a “stratified application”. It applies an entire level of the type stratification. Because of type homogeneity, sufficiently many terms are applied to make the order of the term decrease, so no side-condition is necessary. (b) **(h-app_{partial})** is a partial application: it applies only two term together provided that some condition on types is verified;
- (v) Type-homogeneity constrains the order in which the variables are abstracted: in the **(h-abs)** rule, if a variable of a given order is abstracted then all the lower layers in the stratified context need to be abstracted as well;

- (vi) Because of the previous point and because contexts are stratified, the side-condition present in the rule (abs) of the original safe lambda calculus is always verified and is not required here. Instead the side-condition in (h-abs) ensures that the type $(\overline{A_n}|B)$ is homogeneous.

Lemma 3.1.18 (Basic properties). *Let $\Gamma \vdash_h M : B$ be a valid judgment then*

- (i) B is homogeneous;
- (ii) $\forall z : A \in \Gamma : z \in FV(M) \implies \text{ord } A \geq \text{ord } B$;
- (iii) (Context reduction) $\Gamma_M \vdash_h M : B$ where $\Gamma_M = \{z : A \in \Gamma \mid z \in FV(M)\}$.

Proof. (i) and (ii) are proved by a trivial induction. (iii) Variables in Γ not occurring free in M are necessarily introduced by the weakening rule. The derivation of $\Gamma_M \vdash_h M : A$ can be obtained by removing all the unnecessary application of the weakening rule from the derivation tree of $\Gamma \vdash_h M : A$. \square

Proposition 3.1.3. *strat(Γ) $\vdash_h M : T$ (resp. strat(Γ) $\vdash_{h.\text{app}} M : T$) is a valid judgment if and only if there is a derivation tree for $\Gamma \vdash_s M : T$ (resp. $\Gamma \vdash_{\text{app}} M : T$) in the Curry-style safe lambda calculus (Def. 3.1.1) such that all the types appearing in the derivation tree are homogeneously-typed.*

Proof. *Only if:* The proof is by a trivial structural induction on $\Gamma \vdash_h M : T$. *If:* We proceed by structural induction on the derivation tree of $\Gamma \vdash_s M : T$. The cases (var), (const), (wk) and (app_{as}) are trivial. Suppose that the rule (app) is used. Then we can form the equivalent homogeneous term by using the I.H. and applying (app_{strat}) several times followed by one application of (app_{partial}).

Abstraction: The sequent is of the form $\Gamma \vdash_s \lambda x_1 \dots x_n. s : (A_1, \dots, A_n, B)$ with $\text{ord } \Gamma \geq \text{ord } (A_1, \dots, A_n, B)$. By the induction hypothesis we have $\text{strat}(\Gamma, x_1 : A_1, \dots, x_n : A_n) \vdash_{h.\text{app}} s : B$.

Since we have $\text{ord } \Gamma \geq \text{ord } (A_1, \dots, A_n, B)$, all the variables in Γ have order strictly greater than the variables x_1, \dots, x_n . Therefore there exists a stratification of Γ, x_1, \dots, x_n of the form

$$\text{strat}(\Gamma) \mid \overline{y_1} : \overline{Y_1} \mid \dots \mid \overline{y_l} : \overline{Y_l}$$

for some $l \geq 1$ such that the sequence of variables $\overline{y_1}, \dots, \overline{y_l}$ is equal to x_1, \dots, x_n . Hence using the permutation rule (perm) we can form the judgment

$$\text{strat}(\Gamma) \mid \overline{y_1} : \overline{Y_1} \mid \dots \mid \overline{y_l} : \overline{Y_l} \vdash_{h.\text{app}} s : B.$$

We can now apply the rule (h-abs) to form $\text{strat}(\Gamma) \vdash_{h.\text{app}} \lambda x_1 \dots x_n. s : (A_1, \dots, A_n, B)$. The side-condition of the rule is verified because (A_1, \dots, A_n, B) is homogeneous by assumption. \square

Example 3.1.6.

- (i) The untyped term $(\lambda f x. x)gy$ is homogeneously safe. One possible derivation is:

$$\begin{array}{c}
 \text{(var)} \frac{}{x : o \vdash_h x : o} \\
 \text{(\delta)} \frac{}{\vdash_{h.\text{app}} x : o} \\
 \text{(abs)} \frac{}{\vdash_h \lambda x. x : 1} \\
 \text{(wk)} \frac{}{f : (o, o) \vdash_h \lambda x. x : 1} \\
 \text{(abs)} \frac{}{\vdash_h \lambda f x. x : (1, o, o)} \\
 \text{(wk)} \frac{}{g : (o, o) \vdash_h \lambda f x. x : (1, o, o)} \quad \text{(var)} \frac{}{g : 1 \vdash_h g : 1} \\
 \text{(app}_{\text{strat}}) \frac{}{g : 1 \vdash_h (\lambda f x. x)g : 1} \quad \text{(var)} \frac{}{y : o \vdash_h y : o} \\
 \text{(wk)} \frac{}{g : 1, y : o \vdash_h (\lambda f x. x)g : 1} \quad \text{(wk)} \frac{}{g : 1, y : o \vdash_h y : o} \\
 \text{(app}_{\text{strat}}) \frac{}{g : 1, y : o \vdash_h (\lambda f x. x)gy : o}
 \end{array}$$

(ii) The annotated-terms $\lambda g^{(o,(o,o),o)} x^o. gx$ and $\lambda g^{(o,(o,o),o)} x^o. gx(\lambda x.x)$ are both safe but not homogeneously safe because they are not homogeneously typed. This shows that the safe lambda-calculus strictly contains the homogeneous safe lambda-calculus.

(iii) The annotated-term $\lambda x^0 f^1 \varphi^2. \varphi$ is safe but not homogeneously safe because its type $(0, 1, 2, 2)$ is not homogeneous. On the other hand, the untyped term $\lambda x f \varphi. \varphi$ is homogeneously safe because the annotation $\lambda x^0 f^0 \varphi^0. \varphi$ is safe and homogeneously typed.

Example 3.1.7. The following term is a counter-example used by Sereni [Ser05] to show that not all simply typed terms are *size-change terminating* [LJBA01]:

$$E \equiv (\lambda a. a(\lambda b. a(\lambda c d. d)))(\lambda e. e(\lambda f. f))$$

Claim: The untyped term E is *universally safe*.

Indeed, let $E' \in \Lambda_{\mathbb{T}}$ be a type-annotation of E (i.e., $|E'| = E$) such that E' is typable in the Church simply typed lambda calculus. Then it is easy to check that we have

$$\vdash_{\text{Ch}} E' : A \rightarrow A$$

for some type $A \in \mathbb{T}$ (and thus E has for principal type $\alpha \rightarrow \alpha$) and the type assignments for the bound variables in E' are of the form:

$$\begin{aligned} a &: C \rightarrow A \rightarrow A \\ b &: B \rightarrow B \\ c &: B \rightarrow B \\ d &: A \\ e &: C \equiv (B \rightarrow B) \rightarrow A \rightarrow A \\ f &: B \end{aligned}$$

for some for some types $A, B \in \mathbb{T}$ (not necessarily atomic). It is then an easy exercise to check that for any type $A, B \in \mathbb{T}$, we can form the following term-in-context:

$$\vdash_s E' : A \rightarrow A .$$

On the other hand, E is only homogeneously safe (and not universally homogeneously safe). More precisely, its annotation E' is *homogeneously safe* if and only if $\text{ord } B \geq \text{ord } A - 1$. Formally:

$$\vdash_h E' : A \rightarrow A \quad \Longleftrightarrow \quad \text{ord } B \geq \text{ord } A - 1 .$$

(In particular, the condition in the right-hand side implies that A, B and the types of a, b, c, d, e, f are all homogeneous.)

Related work: In her thesis, de Miranda proposed a different notion of safe lambda calculus [dM06]. This notion corresponds to (a less general version of) our notion of *homogeneous* safe lambda calculus: It can be shown that the applicative fragment (i.e., without lambda-abstraction) of de Miranda's typing system coincides with the applicative fragment of the system of Def. 3.1.10. In particular a version of Proposition 3.1.1 is shown by de Miranda [dM06]. In the presence of lambda abstraction, however, our system is less restrictive. For instance the judgment $\vdash_h \lambda f^{(o,o,o)} x^o. fx : (o, o)$ is derivable in the homogeneous safe lambda calculus but not in the safe lambda calculus *à la* de Miranda. One can show that the system introduced by de Miranda is in fact equivalent to the fragment of the *long-safe lambda calculus* (Def. 3.1.8) restricted to homogeneous types.

3.2 Complexity

This section is concerned with the complexity of the beta-eta equivalence problem for the safe lambda calculus: given two safe lambda terms, are they equivalent up to $\beta\eta$ -conversion?

Let $\exp_h(m)$ denote the *tower of exponential* function defined by:

$$\begin{aligned}\exp_0(m) &= m \\ \exp_{h+1}(m) &= 2^{\exp_h(m)}\end{aligned}$$

Recall that a program is *elementary recursive* if its run-time can be bounded by $\exp_K(n)$ for some constant K where n is the length of the input.

3.2.1 Statman's result

A famous result by Statman states that deciding the $\beta\eta$ -equality of two first-order typable lambda terms is not elementary recursive [Sta79b]. The proof proceeds by encoding the Henkin quantifier elimination of type theory in the simply typed lambda calculus. Simpler proofs have subsequently been given: one by Mairson [Mai92] and another by Loader [Loa98a]. Both proceed by encoding the Henkin quantifier elimination procedure in the lambda calculus, as in the original proof, but their use of list iteration to perform quantifier elimination makes them much easier to understand.

It turns out that all these encodings rely on unsafe terms: Statman's encoding uses the conditional function **sg** which is not definable in the safe lambda calculus [BO07]; Mairson's encoding uses unsafe terms to encode both quantifier elimination and set membership, and Loader's encoding uses unsafe term to build list iterators. We are thus led to conjecture that finite type theory (see definition in Sec. 3.2.2) is intrinsically unsafe in the sense that every encoding of it in the lambda calculus is necessarily unsafe. Of course this conjecture does not rule out the possibility that another non-elementary problem is encodable in the safe lambda calculus.

We start this section by presenting an adaptation of Mairson's encoding. We show that quantifier elimination can be safely encoded and explain why it is problematic to encode set-membership safely. We will then use this encoding to interpret the True Quantifier Boolean Formula (TQBF) problem in the safe lambda calculus, thus showing that deciding beta-eta equality is PSPACE-hard.

3.2.2 Mairson's encoding

We recall the definition of finite type theory. We define $\mathcal{D}_0 = \{\mathbf{true}, \mathbf{false}\}$ and $\mathcal{D}_{k+1} = \text{powerset}(\mathcal{D}_k)$. For $k \geq 0$, we write x^k , y^k and z^k to denote variables ranging over \mathcal{D}_k . Prime formulas are x^0 , $\mathbf{true} \in y^1$, $\mathbf{false} \in y^1$, and $x^k \in y^{k+1}$. Formulae are built up from prime formulae using the logical connectives $\wedge, \vee, \rightarrow, \neg$ and the quantifiers \forall and \exists . Meyer showed that deciding the validity of such formulae requires nonelementary time [Mey74].

In Mairson's encoding, boolean values are encoded by terms of type $\mathbf{B} = \sigma \rightarrow \sigma \rightarrow \sigma$ for some type σ , and variables of order $k \geq 0$ are encoded by terms of type Δ_k defined as $\Delta_0 \equiv \mathbf{B}$ and $\Delta_{k+1} \equiv \Delta_k^*$ where for any type α , $\alpha^* = (\alpha \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$ for some type τ . Using this encoding, unsafety manifests itself in two different ways.

1. First in the encoding of set membership. The prime formula $x^k \in y^{k+1}$ is encoded as

$$x^k : \Delta_k, y^{k+1} : \Delta_{k+1} \vdash_{\text{st}} y^{k+1}(\lambda y^k : \Delta_k. \text{OR}(\text{eq}_k \underline{x^k} y^k)) F : \Delta_k \rightarrow \Delta_{k+1} \rightarrow \Delta_0 \quad (3.3)$$

for some terms OR , F , eq_k . This term is unsafe because of the underline occurrence of x^k which is not abstracted together with y^k .

2. Secondly, quantifier elimination is performed using a list iterator \mathbf{D}_{k+1} of type Δ_{k+2} which acts like the *fold_right* function from functional programming over the list of all elements of \mathcal{D}_k . Thus for instance the formula $\forall x^0. \exists y^0. x^0 \vee y^0$ is encoded as

$$\vdash_{\text{st}} \mathbf{D}_0(\lambda x^0 : \Delta_0. \text{AND}(\mathbf{D}_0(\lambda y^0 : \Delta_0. \text{OR}(\underline{x^0} \vee y^0))F)) T : \mathbf{B}$$

where the type τ is instantiated as \mathbf{B} . This term is unsafe since the underlined occurrence is unsafely bound. This is due to the presence of two nested quantifiers in the formula, which are

encoded as two nested list iterations. More generally, nested binding will be encoded safely if and only if every variable x in the formula is bound by the first quantifier $\exists z$ or $\forall z$ in the path to the root of the AST of the formula verifying $\text{ord } z \geq \text{ord } x$. For instance if set-membership could be encoded safely then the interpretation of $\forall x^k. \exists y^{k+1}. x^k \in y^{k+1}$ would be unsafe whereas the encoding of $\forall y^{k+1}. \exists x^k. x^k \in y^{k+1}$ would be safe.

Surprisingly, the ‘unsafety’ of the quantifier elimination procedure can be easily overcome. The idea is as follows. We introduce multiple domains of representation for a given formula. An element of \mathcal{D}_k is thereby represented by countably many terms of type Δ_k^n where $n \in \mathbb{N}$ indicates the level of the domain of representation. The type Δ_k^n is defined in such a way that its order strictly increases as n grows. Furthermore, there exists a term that can lower the domain of representation of a given term. Thus each formula variable can have a different domain of representation, and since there are infinitely many such domains, it is always possible to find an assignment of representation domains to variables such that the resulting encoding term is safe.

For set-membership, however, there is no obvious way to obtain a safe encoding. In order to turn Mairson’s encoding of set-membership (Eq. 3.3) into a safe term, we would need to have access to a function that changes the domain of representation of an encoded higher-order value of the type-hierarchy. Unfortunately, such transformation is intrinsically unsafe!

We now present the encoding in details.

3.2.2.1 Encoding basic boolean operations

Let o be a base type and define the family of types $\sigma_0 \equiv o$, $\sigma_{n+1} \equiv \sigma_n \rightarrow \sigma_n$ satisfying $\text{ord } \sigma_n = n$. Booleans are encoded over domains $\mathbf{B}_n \equiv \sigma_n \rightarrow o \rightarrow o \rightarrow o$ for $n \geq 0$, each type \mathbf{B}_n being of order $n + 1$. We write $\dot{\mathbf{I}}_{n+1}$ to denote the term $\lambda x^{\sigma_n}. x : \sigma_{n+1}$ for $n \geq 0$. The truth values **true** and **false** are represented by the following terms parameterized by $n \in \mathbb{N}$:

$$\begin{aligned} T^n &\equiv \lambda u^{\sigma_n} x^o y^o. x : \mathbf{B}_n \\ F^n &\equiv \lambda u^{\sigma_n} x^o y^o. y : \mathbf{B}_n \end{aligned}$$

Clearly these terms are safe. Moreover the following relations hold for all $n, n' \geq 0$:

$$\begin{aligned} \lambda u^{\sigma_{n'}}. T^{n+1} \dot{\mathbf{I}}_{n+1} &\rightarrow_{\beta} T^{n'} \\ \lambda u^{\sigma_{n'}}. F^{n+1} \dot{\mathbf{I}}_{n+1} &\rightarrow_{\beta} F^{n'} \end{aligned}$$

Hence it is possible to change the domain of representation of a Boolean value from a higher-level to another arbitrary level using the transformation:

$$\mathbf{C}_0^{n+1 \mapsto n'} \equiv \lambda m^{\mathbf{B}_{n+1}} u^{\sigma_{n'}}. m \dot{\mathbf{I}}_{n+1} : \mathbf{B}_{n+1} \rightarrow \mathbf{B}_{n'}$$

so that if a term $M : \mathbf{B}_n$ for $n \geq 1$ is beta-eta convertible to T^n (resp. F^n) then $\mathbf{C}_0^{n \mapsto n'} M : \mathbf{B}_{n'}$ is beta-eta convertible to $T^{n'}$ (resp. $F^{n'}$).

Observe that although the term $\mathbf{C}_0^{n+1 \mapsto n'}$ is safe for all $n, n' \geq 0$, if we apply it to a variable then the resulting term

$$x : \mathbf{B}_{n+1} \vdash_{\text{st}} \mathbf{C}_0^{n+1 \mapsto n'} x : \mathbf{B}_n$$

is safe if and only if the transformation decreases the domain of representation of x (i.e., $\text{ord } \mathbf{B}_{n+1} \geq \text{ord } \mathbf{B}_{n'}$).

Boolean functions are encoded by the following safe terms parameterized by n :

$$\begin{aligned} AND^n &\equiv \lambda p^{\mathbf{B}_n} q^{\mathbf{B}_n} u^{\sigma_n} x^o y^o. p \ u \ (q \ u \ x \ y) \ y : \mathbf{B}_n \rightarrow \mathbf{B}_n \rightarrow \mathbf{B}_n \\ OR^n &\equiv \lambda p^{\mathbf{B}_n} q^{\mathbf{B}_n} u^{\sigma_n} x^o y^o. p \ u \ x \ (q \ u \ x \ y) : \mathbf{B}_n \rightarrow \mathbf{B}_n \rightarrow \mathbf{B}_n \\ NOT^n &\equiv \lambda p^{\mathbf{B}_n} u^{\sigma_n} x^o \lambda y^o. p \ u \ y \ x : \mathbf{B}_n \rightarrow \mathbf{B}_n \rightarrow \mathbf{B}_n \end{aligned}$$

3.2.2.2 Coding elements of the type hierarchy

For any $n \in \mathbb{N}$ we define the hierarchy of type Δ_k^n as follows: $\Delta_0^n \equiv B_n$ and $\Delta_{k+1}^n \equiv \Delta_k^{n*}$ where for any type α , $\alpha^* = (\alpha \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$. An occurrence of a formula variable x^k will be encoded as a term variable x^k of type Δ_k^n for some level of domain representation $n \in \mathbb{N}$. Following Mairson's encoding, each set \mathcal{D}_k is represented by a list \mathbf{D}_k^n constituted of all its elements:

$$\begin{aligned} \mathbf{D}_0^n &\equiv \lambda c^{B_n \rightarrow \tau \rightarrow \tau} e^\tau . c \ T^n \ (c \ F^n \ e) : \Delta_1^n \\ \mathbf{D}_{k+1}^n &\equiv \text{powerset } \mathbf{D}_k^n : \Delta_{k+2}^n \end{aligned}$$

where

$$\begin{aligned} \text{powerset} &\equiv \lambda A^{*(\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**}} . \\ &\quad A^* \ \text{double} \ (\lambda c^{\alpha^* \rightarrow \tau \rightarrow \tau} b^\tau . c \ (\lambda c'^{\alpha \rightarrow \tau \rightarrow \tau} b'^\tau . b') \ b) \\ &\quad : ((\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \\ \text{double} &\equiv \lambda x^\alpha \ l^{(\alpha^* \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau} c^{\alpha^* \rightarrow \tau \rightarrow \tau} b^\tau . \\ &\quad l(\lambda e^{\alpha^*} . c \ (\lambda c'^{\alpha \rightarrow \tau \rightarrow \tau} b'^\tau . c' \ \underline{x} \ (e \ c' \ b')))(l \ c \ b) \\ &\quad : \alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**} . \end{aligned}$$

In all these terms, the only variable occurrence that is potentially unsafe is the underlined occurrence x in *double*. This occurrence is safely bound just when $\text{ord } \alpha \geq \text{ord } \tau$. Consequently for all $k, n \geq 0$, \mathbf{D}_k^n is safe if and only if $\text{ord } \alpha \geq \text{ord } \tau$.

3.2.2.3 Quantifier elimination

Terms of type Δ_{k+1}^n are now used as iterators over list of elements of type Δ_k^n and we set $\tau \equiv B_n$ in the type Δ_{k+1}^n in order to iterate a level- n Boolean function. Since $\text{ord } \Delta_k^n \geq \text{ord } B_n$ for all n , all the instantiations of the terms \mathbf{D}_k^n will be safe. Following [Mai92], quantifier elimination interprets the formula $\forall x^k. \Phi(x^k)$ as the iterated conjunction: $\mathbf{C}_0^{n \mapsto 0} \left(\mathbf{D}_k^n (\lambda x^k : \Delta_k^n . \text{AND}^n (\hat{\Phi} \ x^k)) \ T^n \right)$ where $\hat{\Phi}$ is the interpretation of Φ and n is the representation level chosen for the variable x^k ; similarly $\exists x^k. \Phi(x^k)$ is interpreted by the iterated disjunction $\mathbf{C}_0^{n \mapsto 0} \left(\mathbf{D}_k^n (\lambda x^k : \Delta_k^n . \text{AND}^n (\hat{\Phi} \ x^k)) \ T^n \right)$.

Let $x_p^{k_p} \dots x_1^{k_1}$ for $p \geq 1$ be the list of variables appearing in the formula. W.l.o.g. we can assume that they are given in the order of appearance of their binder in the formula (*i.e.*, $x_p^{k_p}$ is bound by the leftmost binder). We fix the domain of representation of each variable as follows. The right-most variable $x_1^{k_1}$ will be encoded in the domain $\Delta_{k_1}^0$; suppose that for $1 \leq i < p$ the domain of representation of $x_i^{k_i}$ is $\Delta_{k_i}^l$ then the domain of representation of $x_{i+1}^{k_{i+1}}$ is defined as $\Delta_{k_{i+1}}^{l'}$ where l' is the smallest natural number such that $\text{ord } \Delta_{k_{i+1}}^{l'}$ is strictly greater than $\text{ord } \Delta_{k_i}^l$.

This way, since variables that are bound first have higher order, the variables that are bound in the nested list-iterations (corresponding to the nested quantifiers in the formula) are necessarily safely bound.

Example 3.2.1. The formula $\forall x^0. \exists y^0. x^0 \vee y^0$, which is encoded by an unsafe term in Mairson's encoding, is represented in our encoding by the safe term:

$$\vdash_s \mathbf{C}_0^{1 \mapsto 0} \left(\mathbf{D}_0^1 (\lambda x^0 : \Delta_0^1 . \text{AND}^0 (\mathbf{D}_0^0 (\lambda y^0 : \Delta_0^0 . \text{OR}^0 (\text{OR}^0 (\mathbf{C}_0^{1 \mapsto 0} \ x^0) \ y^0)) \ F^0)) \ T^1 \right) : B_0 .$$

3.2.2.4 Set-membership

To complete the interpretation of prime formulae, we would need to show how to encode set membership. The use of multiple domains of representation does not suffice to turn Mairson's encoding into a safe term. We would further need to have a version of the Booleans conversion

term $\mathbf{C}_0^{n+1 \mapsto n'}$ generalized to higher-order sets. This transformation can be interpreted as the simply typed term:

$$\mathbf{C}_{k+1}^{n \mapsto n'} \equiv \lambda m^{\Delta_{k+1}^n} u^{\Delta_k^n \rightarrow \tau \rightarrow \tau} v^\tau . m(\lambda z^{\Delta_k^n} w^\tau . \underline{u(\mathbf{C}_k^{n \mapsto n'} z)w})v : \Delta_{k+1}^n \rightarrow \Delta_{k+1}^{n'} \text{ enspace.}$$

Unfortunately this term is safe if and only if $n = n'$ —the largest underlined subterm is safe just when $n \geq n'$ and the other underline subterm is safe just when $n' \geq n$ —in which case the transformation is of no interest.

This leads us to conjecture that the set-membership test function is intrinsically unsafe.

If $\mathbf{C}_{k+1}^{n \mapsto n'}$ were safely representable then the encoding would go as follows: We set $\tau \equiv \mathbf{B}_0$ in the types Δ_{k+1}^n for all $n, k \geq 0$ in order to iterate a level-0 Boolean function. Firstly, the formulae “**true** $\in y^1$ ” and “**false** $\in y^1$ ” can be encoded by the safe terms $y^1(\lambda x^0. \text{OR}^0 x^0)F^0$ and $y^1(\lambda x^0. \text{OR}^0(\text{NOT}^0 x^0))F^0$ respectively. For the general case “ $x^k \in y^{k+1}$ ” we proceed as in Mairson’s proof [Mai92]: we introduce lambda-terms encoding set equality, set membership and subset tests; we further parameterize these encoding by a natural number n .

$$\begin{aligned} \text{member}_{k+1}^{n+1} &\equiv \lambda x^{\Delta_k^{n+1}} y^{\Delta_{k+1}^{n+1}} . (\mathbf{C}_{k+1}^{n+1 \mapsto n} y) (\lambda z^{\Delta_k^n} . \text{OR}^0 (eq_k^n (\mathbf{C}_k^{n+1 \mapsto n} x) z)) F^0 \\ &\quad : \Delta_k^{n+1} \rightarrow \Delta_{k+1}^{n+1} \rightarrow \mathbf{B}_0 \\ \text{subset}_{k+1}^n &\equiv \lambda x^{\Delta_{k+1}^n} y^{\Delta_{k+1}^n} . x (\lambda x^{\Delta_k^n} . \text{AND}^0 (\text{member}_{k+1}^n x y)) T^0 \\ &\quad : \Delta_{k+1}^n \rightarrow \Delta_{k+1}^n \rightarrow \mathbf{B}_0 \\ eq_0^n &\equiv \lambda x^{\mathbf{B}_n} . \lambda y^{\mathbf{B}_n} . \mathbf{C}_0^{n \mapsto 0} (\text{OR}^n (\text{AND}^n x y) (\text{AND}^n (\text{NOT}^n x) (\text{NOT}^n y))) \\ &\quad : \mathbf{B}_n \rightarrow \mathbf{B}_n \rightarrow \mathbf{B}_0 \\ eq_{k+1}^n &\equiv \lambda x^{\Delta_{k+1}^n} y^{\Delta_{k+1}^n} . (\lambda op^{\Delta_{k+1}^n \rightarrow \Delta_{k+1}^n \rightarrow \mathbf{B}_0} . \text{AND}^0 (op x y) (op y x)) \text{subset}_{k+1}^n \\ &\quad : \Delta_{k+1}^n \rightarrow \Delta_{k+1}^n \rightarrow \mathbf{B}_0 . \end{aligned}$$

The variables in the definition of eq_{k+1}^n and subset_{k+1}^n are safely bounds. Moreover, the occurrence of x in $\text{member}_{k+1}^{n+1}$ is now safely bound (this was not the case in Mairson’s original encoding) thanks to the fact that the representation domain of z is lower than that of x . The formula $x^k \in y^{k+1}$ can then be encoded as

$$x : \Delta_k^n, y : \Delta_{k+1}^{n'} \vdash_{\text{st}} \text{member}_{k+1}^u (\mathbf{C}_k^{n \mapsto u} x) (\mathbf{C}_{k+1}^{n' \mapsto u} y) : \mathbf{B}_0$$

for some $n, n' \geq 2$ and $u = \min(n, n') + 1$.

Unfortunately, this encoding is not completely safe because it uses the unsafe conversion terms $\mathbf{C}_k^{n \mapsto n'}$ for $k \geq 1$.

3.2.3 PSPACE-hardness

We observe that instances of the True Quantified Boolean Formulae satisfaction problem (TQBF) are special instances of the decision problem for finite type theory. These instances corresponds to formulae in which set membership is not allowed and variables are all taken from the base domain \mathcal{D}_0 . As we have shown in the previous section, such restricted formulae can be safely encoded in the safe lambda calculus. Therefore since TQBF is PSPACE-complete, deciding $\beta\eta$ -equality of two terms of the safe lambda calculus is PSPACE-hard.

Example 3.2.2. Using the encoding where τ is set to \mathbf{B}_0 in the types Δ_k^n for all $k, n \geq 0$, the

formula $\forall x \exists y \exists z (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$ is represented by the safe term:

$$\begin{aligned}
& \vdash_s \mathbf{D}_0^2(\lambda x^{\mathbf{B}_2}. \mathbf{AND}^0 \\
& \quad (\mathbf{D}_0^1(\lambda y^{\mathbf{B}_1}. \mathbf{OR}^0 \\
& \quad \quad (\mathbf{D}_0^0(\lambda z^{\mathbf{B}_0}. \mathbf{OR}^0 \\
& \quad \quad \quad (\mathbf{AND}^0(\mathbf{OR}^0(\mathbf{OR}^0(\mathbf{C}_0^{2 \mapsto 0} x) (\mathbf{C}_0^{1 \mapsto 0} y))z) \\
& \quad \quad \quad \quad (\mathbf{OR}^0(\mathbf{OR}^0(\mathbf{NEG}^0(\mathbf{C}_0^{2 \mapsto 0} x))(\mathbf{NEG}^0(\mathbf{C}_0^{1 \mapsto 0} y)))(\mathbf{NEG}^0 z))) \\
& \quad \quad \quad \quad \quad \mathbf{F}^0) \\
& \quad \quad \quad \quad \quad \mathbf{F}^0) \\
& \quad \quad \quad \quad \mathbf{T}^0 \\
& \quad \quad \mathbf{B}_0 \text{ .}
\end{aligned}$$

REMARK 3.2.1 Since the Boolean satisfaction problem (SAT) can be reduced to TQBF (where formulae are restricted to use only existential quantifiers), the safe lambda calculus is also NP-hard. Asperti gave an interpretation of SAT in the simply typed lambda calculus but his encoding relies on unsafe terms [Asp].

3.2.4 Other complexity results

3.2.4.1 Better lower bound?

Since the safety condition restricts the expressivity of the lambda calculus in a non-trivial way, one can reasonably expect the beta-eta equality problem (where types are not restricted) to have a lower complexity in the safe case than in the normal case. Our failed attempt to encode type theory in the safe lambda calculus suggests that the non-elementary lower bounds that holds in the simply typed lambda calculus no longer applies in the safe lambda calculus. (Nevertheless, one may not rule out the possibility that another non recursive problem may be encodable in the safe lambda calculus.)

We have shown that the problem is PSPACE-hard but this is probably a coarse lower bound. It would be interesting to know whether it is also EXPTIME-hard.

3.2.4.2 Upper bound

At present, no upper bound is known for the equivalence problem for safe terms.

3.2.4.3 Beta-eta equivalence for terms limited to a finite set of types

Statman showed [Sta79b] that there exists a finite set of types such that the beta-eta equivalence problem restricted to terms of these types is PSPACE-hard.

The picture is different in the safe lambda calculus since our encoding of TQBF requires the full type hierarchy. It was indeed necessary to introduce variables of higher-order in order to eliminate ‘unsafety’. Consequently, we had to use simple types of unbounded order (the order is linear in the size of the QBF formula). We suspect the decidability problem for safe terms restricted to any finite set of types to have a complexity lower than PSPACE.

3.2.4.4 Normalization

The *normalization problem* is: given a term M , what is its β -normal form? This problem is non-elementary even when restricted to safe terms as the following example shows. Let $\tau_{-2} \equiv o$ and for $n \geq -1$, $\tau_n \equiv \tau_{n-1} \rightarrow \tau_{n-1}$. For $k, n \in \mathbb{N}$ we write \overline{k}^n to denote the k th Church Numeral parameterized by n as follows:

$$\overline{k}^n \equiv \lambda s^{\tau_{n-1}} z^{\tau_{n-2}}. \overbrace{s(\dots(s(s z) \dots))}^{k \text{ times}} : \tau_n \text{ .}$$

Then for $n \geq 1$, the safe term $\overline{2}^{n-1} \overline{2}^{n-2} \dots \overline{2}^0$ of type τ_0 has length $\mathcal{O}(n)$ whereas its normal form $\overline{\exp_n(1)}^0$ has length $\mathcal{O}(\exp_n(1))$.

Statman's result shows that in the simply typed lambda calculus, the beta-eta equality problem is essentially as hard as the normalization problem: they are both non-elementary. It is not known whether this is still the case in the safe lambda calculus. In particular, it may be the case that the beta-eta equivalence problem is elementary although we know that the normalization problem is not.

3.2.4.5 The beta-reduction problem

The *beta-reduction problem* is related to the beta-eta equivalence problem. It can be stated as follows: given a term M_1 in β -normal form and a term M_2 (possibly containing redexes), does M_2 β -reduce to M_1 ?

Schubert gave a PSPACE algorithm to decide the β -reduction problem for order-3 lambda terms [Sch01]. Since order-3 terms are sufficient to encode TQBF in the lambda calculus, Schubert shows that the problem is PSPACE-complete. No complexity result is known for restrictions of this problem to terms of order greater than 3. A natural question to ask is whether complexity characterizations can be obtained when restricting the problem to safe terms.

3.3 Expressivity

3.3.1 Numeric functions representable in the safe lambda calculus

Natural numbers can be encoded in the simply typed lambda calculus using the Church Numerals: each $n \in \mathbb{N}$ is encoded as the term $\overline{n} = \lambda sz. s^n z$ of type $I = ((o, o), o, o)$ where o is a ground type. We say that a p -ary function $f : \mathbb{N}^p \rightarrow \mathbb{N}$, for $p \geq 0$, is represented by a term $F : (I, \dots, I, I)$ (with $p + 1$ occurrences of I) if for all $m_i \in \mathbb{N}$, $0 \leq i \leq p$ we have:

$$F \overline{m_1} \dots \overline{m_p} =_\beta \overline{f(m_1, \dots, m_p)}.$$

In 1976 Schwichtenberg [Sch76] showed the following:

Theorem 3.3.1 (Schwichtenberg 1976). *The numeric functions representable by simply-typed lambda-terms of type $I \rightarrow \dots \rightarrow I$ using the Church Numeral encoding are exactly the multivariate polynomials extended with the conditional function.*

If we restrict ourselves to safe terms, the representable functions are exactly the multivariate polynomials:

Theorem 3.3.2. *The functions representable by safe λ -expressions of type $I \rightarrow \dots \rightarrow I$ are exactly the multivariate polynomials.*

Proof. Natural numbers are encoded as the Church Numerals: $\overline{n} = \lambda sz. s^n z$ for each $n \in \mathbb{N}$. Addition: For $n, m \in \mathbb{N}$, $\overline{n + m} = \lambda \alpha^{(o, o)} x^o. (\overline{n} \alpha) (\overline{m} \alpha x)$. Multiplication: $\overline{n \cdot m} = \lambda \alpha^{(o, o)}. \overline{n} (\overline{m} \alpha)$. These terms are safe and clearly any multivariate polynomial $P(n_1, \dots, n_k)$ can be computed by composing the addition and multiplication terms as appropriate.

For the converse, let U be a safe lambda-term of type $I \rightarrow I \rightarrow I$. The generalization to terms of type $I^n \rightarrow I$ for any $n \in \mathbb{N}$ is immediate (they correspond to polynomials with n variables). By Lemma 3.1.2, safety is preserved by η -long normal expansion therefore we can assume that U is in η -long normal form.

Let \mathcal{N}_Σ^τ denote the set of safe η -long β -normal terms of type τ with free variables in Σ , and \mathcal{A}_Σ^τ for the set of β -normal terms of type τ with free variables in Σ and of the form $\varphi s_1 \dots s_m$ for some variable $\varphi : (A_1, \dots, A_m, o)$ where $m \geq 0$ and for all $1 \leq i \leq m$, $s_i \in \mathcal{N}_\Sigma^{A_i}$. Observe that the set \mathcal{A}_Σ^o contains only safe terms but the sets \mathcal{A}_Σ^τ in general may contain unsafe terms. Let Σ

denote the alphabet $\{x, y : I, z : o, \alpha : o \rightarrow o\}$. The sets \mathcal{N}_\emptyset^0 is given by the following grammar defined over the set of terminals $\Sigma \cup \{\lambda xy\alpha z., \lambda z.\}$:

$$\begin{aligned} \mathcal{N}_\emptyset^{(I,I,I)} &\rightarrow \lambda xy\alpha z. \mathcal{A}_\Sigma^o \\ \mathcal{A}_\Sigma^o &\rightarrow z \mid \mathcal{A}_\Sigma^{(o,o)} \mathcal{A}_\Sigma^o \\ \mathcal{A}_\Sigma^{(o,o)} &\rightarrow \alpha \mid \mathcal{A}_\Sigma^I \mathcal{N}_\Sigma^{(o,o)} \\ \mathcal{N}_\Sigma^{(o,o)} &\rightarrow \lambda z. \mathcal{A}_\Sigma^o \\ \mathcal{A}_\Sigma^I &\rightarrow x \mid y . \end{aligned}$$

The key rule is the fourth one: if we had not imposed the safety constraint the right-hand side would instead be of the form $\lambda w^o. \mathcal{A}_{\Sigma \cup \{w:o\}}^{(o,o)}$. Here the safety constraint imposes to abstract all the ground type variables occurring freely, thus only one free variable of ground type can appear in the term and we can choose it to be named z up to α -conversion.

We extend the notion of representability to terms of type o , (o, o) and I with free variables in Σ as follows: a function $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ is represented by a term $\Sigma \vdash_{\text{st}} F : o$ if and only if for all $m, n \in \mathbb{N}$, $F[\overline{m}, \overline{n}/x, y] =_\beta \alpha^{\overline{f(m,n)}} z$, by a term $\Sigma \vdash_{\text{st}} G : (o, o)$ iff $G[\overline{m}, \overline{n}/x, y] =_\beta \lambda z. \alpha^{\overline{f(m,n)}} z$, and by $\Sigma \vdash_{\text{st}} H : I$ iff $H[\overline{m}, \overline{n}/x, y] =_\beta \lambda \alpha z. \alpha^{\overline{f(m,n)}} z$.

We now show by induction on the grammar rules that any term generated by the grammar represents some polynomial: The term x and y represent the projection functions $(m, n) \mapsto m$ and $(m, n) \mapsto n$ respectively. The term α and z represent the constant functions $(m, n) \mapsto 1$ and $(m, n) \mapsto 0$ respectively. If $F \in \mathcal{A}_\Sigma^o$ represents the functions f then so does $\lambda z. F$.

We make the following observations: let $m, p, p' \geq 0$

1. $\overline{m}(\lambda z. \alpha^p z) =_\beta \lambda z. \alpha^{m+p} z$
2. $(\lambda z. \alpha^p z)(\alpha^{p'} z) =_\beta \alpha^{p+p'} z$.

Now suppose that $F \in \mathcal{A}_\Sigma^I$ and $G \in \mathcal{N}_\Sigma^{(o,o)}$ represent the functions f and g respectively then by the previous observation, FG represents the function $f \times g$. And if $F \in \mathcal{A}_\Sigma^{(o,o)}$ and $G \in \mathcal{N}_\Sigma^o$ represent the functions f and g then FG represents the function $f + g$.

Thus U represents some polynomial as required: for all $m, n \in \mathbb{N}$ we have $U \overline{m} \overline{n} =_\beta \lambda \alpha z. \alpha^{p(m,n)} z$ where $p(m, n) = \sum_{0 \leq k \leq d} m^{i_k} n^{j_k}$ for some $i_k, j_k \geq 0$, $d \geq 0$. \square

Corollary 3.3.3. *The conditional operator $C : I \rightarrow I \rightarrow I \rightarrow I$ satisfying:*

$$C \ t \ y \ z \rightarrow_\beta \begin{cases} y, & \text{if } t \rightarrow_\beta \overline{0}; \\ z, & \text{if } t \rightarrow_\beta \overline{n+1}. \end{cases}$$

is not definable in the simply-typed safe lambda calculus.

Example 3.3.1. The term $\lambda FGH\alpha x. F(\lambda y. G\alpha x)(H\alpha x)$ used by Schwichtenberg [Sch76] to define the conditional operator is unsafe since the underlined subterm, which is of order 1, occurs at an operand position and contains an occurrence of x of order 0.

REMARK 3.3.1

1. This corollary tells us that the conditional function is not definable when numbers are represented by the Church Numerals. It may still be possible, however, to represent the conditional function using a different encoding for natural numbers. A possible way to compensate for the loss of expressivity caused by the safety constraint consists in introducing countably many domains of representation for natural numbers. This is the technique that is used to represent the predecessor function in the simply typed lambda calculus [FLO83].

2. There are other ways to interpret conditional in the lambda calculus. For instance the (unsafe) lambda term $\lambda txy.(C\ t\ \overline{0}\ \overline{1})(\lambda u.y)x$ of type $I \rightarrow o \rightarrow o \rightarrow o$ behaves like the conditional operator C . It can be shown that there is no such term in the safe lambda calculus simply because the only safe terms of type $I \rightarrow o \rightarrow o \rightarrow o$ up to $\alpha\beta\eta$ -equivalence are $\lambda txy.x$ and $\lambda txy.y$.
3. The boolean conditional can be represented in the safe lambda calculus as follows: We encode booleans by terms of type $B = ((o, o), o, o)$. The two truth values are then represented by $\lambda x^o y^o.x$ and $\lambda x^o y^o.y$ and the conditional by $\lambda F^B G^B H^B.F\ G\ H$.
4. It is also possible to define a conditional operator behaving like the conditional operator C in the second-order lambda calculus [FLO83]: natural numbers are represented by terms $\overline{n} \equiv \Lambda t.\lambda s^{t \rightarrow t}.s^n(z)$ of type $J \equiv \Delta t.(t \rightarrow t) \rightarrow (t \rightarrow t)$ and the conditional is encoded by the term $\lambda F^J G^J H^J.F\ J\ (\lambda u^J.G)\ H$. Whether this term is safe or not cannot be answered just yet as we do not have a notion of safety for second-order typed term.

3.3.2 Word functions definable in the safe lambda calculus.

Schwichtenberg's result on numeric functions definable in the lambda calculus was extended to richer structures: Zaionc studied the problem for words functions, then functions over trees and eventually the general case of functions over free algebras [Lei93, Zai91, Zai88, Zai87, Zai95]. In this section we consider the case of word functions expressible in the safe lambda calculus.

We consider equality of terms modulo α , β and η conversion, and we write $M =_{\beta\eta} N$ to denote this equality. For any simple type τ , we write $\text{Cl}(\tau)$ for the set of closed terms of type τ (modulo α , β and η conversion). We consider a binary alphabet $\Sigma = \{a, b\}$. The result naturally extends to all finite alphabets. We consider the set Σ^* of all words over Σ . The empty word ϵ is denoted ϵ . We write $|w|$ to denote the length of the word $w \in \Sigma^*$. For any $k \in \mathbb{N}$ we write \mathbf{k} to denote the word $a \dots a$ with k occurrences of a , so that $|\mathbf{k}| = k$. For any $n \geq 1$ and $k \geq 0$, we write $c(n, k)$ for the n -ary function $(\Sigma^*)^n \rightarrow \Sigma^*$ that maps all inputs to the word \mathbf{k} . The function $\mathbf{app} : (\Sigma^*)^2 \rightarrow \Sigma^*$ is the usual concatenation function: $\mathbf{app}(x, y)$ is the word obtained by concatenating x and y . The substitution function $\mathbf{sub} : (\Sigma^*)^3 \rightarrow \Sigma^*$ is defined as follows: $\mathbf{sub}(x, y, z)$ is the word obtained from x by substituting the word y for all occurrences of a and z for all occurrences of b .

Take the type $\mathbf{B} = (o \rightarrow o) \rightarrow (o \rightarrow o) \rightarrow o \rightarrow o$, called *the binary word type* [Zai87]. There is a 1-1 correspondence between words over Σ and closed terms of type \mathbf{B} : the empty word ϵ is represented by $\lambda uvx.x$, and if $w \in \Sigma^*$ is represented by a term $W \in \text{Cl}(\mathbf{B})$ then $a \cdot w$ is represented by $\lambda uvx.u(Wuvx)$ and $b \cdot w$ is represented by $\lambda uvx.v(Wuvx)$. The term representing the word w is denoted by \underline{w} . A closed term of type $\mathbf{B}^n \rightarrow \mathbf{B}$ is called a **word function**. We say that the function on words $h : (\Sigma^*)^n \rightarrow \Sigma^*$ is **represented** by the term $H \in \text{Cl}(\mathbf{B}^n \rightarrow \mathbf{B})$ just if for all $x_1, \dots, x_n \in \mathbf{B}^*$, $H \underline{x_1} \dots \underline{x_n} = \underline{hx_1 \dots x_n}$.

Zaionc showed that there exists a finite base of word functions in the sense that every λ -definable word function is some composition of functions from the base [Zai87]:

Theorem 3.3.4 (Zaionc [Zai87]). *The set of λ -definable word functions is the minimal set containing the following word functions and closed by compositions:*

- *concatenation* \mathbf{app} ;
- *substitution* \mathbf{sub} ;
- *extraction of the maximal prefix containing only a given letter*;
- *non-emptiness check*: returns $\mathbf{0}$ if the word is ϵ and $\mathbf{1}$ otherwise, as well as *emptiness check*;
- *occurrence checking*: returns $\mathbf{1}$ if the word contain an occurrence of a given letter and $\mathbf{0}$ otherwise;

- *first-occurrence check: test whether the word begins with a given letter;*
- *all the projections;*
- *all the constant functions.*

The lambda terms representing the base functions are:

$$\begin{aligned}
\text{APP} &= \lambda cduvx.cuv(duvx) & \text{SUB} &= \lambda xdeuvx.c(\lambda y.duvy)(\lambda y.euvy)x \\
\text{CUT}_a &= \lambda cuvxx.cu(\lambda y.x)x & \text{CUT}_b &= \lambda cuvxx.c(\lambda y.x)vx \\
\text{SQ} &= \lambda cuvxx.c(\lambda y.ux)(\lambda y.ux)x & \overline{\text{SQ}} &= \lambda cuvxx.c(\lambda y.x)(\lambda y.x)(ux) \\
\text{BEG}_a &= \lambda cuvxx.c(\lambda y.ux)(\lambda y.x)x & \text{BEG}_b &= \lambda cuvxx.c(\lambda y.x)(\lambda y.ux)x \\
\text{OCC}_a &= \lambda cuvxx.c(\lambda y.ux)(\lambda y.y)x & \text{OCC}_b &= \lambda cuvxx.c(\lambda y.y)(\lambda y.ux)x .
\end{aligned}$$

where APP represents concatenation, SUB substitution, SQ and $\overline{\text{SQ}}$ non-emptiness and emptiness checking, BEG_a and BEG_b first-occurrence test, and OCC_a and OCC_b occurrence test.

We observe that among these terms only APP and SUB are safe. All the other terms are unsafe because they contain terms of the form $N(\lambda y.x)$ where x and y are of the same order. It turns out that APP and SUB constitute a base of terms generating all the functions definable in the safe lambda calculus as the following theorem states:

Theorem 3.3.5. *Let $\lambda^{\text{safe}}\text{def}$ denote the minimal set containing the following word functions and closed by compositions:*

- *concatenation app;*
- *substitution sub;*
- *all the projections;*
- *all the constant functions.*

The set of word-functions definable in the safe lambda calculus is precisely $\lambda^{\text{safe}}\text{def}$.

The proof follows the same steps as Zaionc's proof. The first direction is immediate: the terms APP and SUB are safe and represent concatenation and substitution. Projections are represented by safe terms of the form $\lambda x_1 \dots x_n.x_i$ for some $i \in \{1..n\}$, and constant functions by $\lambda x_1 \dots x_n.\underline{w}$ for some $w \in \Sigma^*$. For composition, take a functions $g : (\Sigma^*)^n \rightarrow \Sigma^*$ represented by safe term $G \in \text{Cl}(\mathbf{B}^n \rightarrow \mathbf{B})$ and functions $f_1, \dots, f_n : (\Sigma^*)^p \rightarrow \Sigma^*$ represented by safe terms F_1, \dots, F_n respectively then the function

$$(x_1, \dots, x_p) \mapsto g(f_1(x_1, \dots, x_p), \dots, f_n(x_1, \dots, x_p))$$

is represented by the term $\lambda c_1 \dots x_p.G(F_1 c_1 \dots c_p) \dots (F_n c_1 \dots c_p)$ which is also safe.

To show the other directions we need to introduce some more definitions. We will write $\text{Op}(n, k)$ to denote the set of open terms of the form:

$$c_1 : \mathbf{B}, \dots, c_n : \mathbf{B}, u : (o, o), v : (o, o), x_{k-1} : o, \dots, x_0 : o \vdash_{\text{st}} M : o .$$

Thus we have the following equality (modulo α , β and η conversions) for $n, k \geq 1$:

$$\text{Cl}(\tau(n, k)) = \{\lambda c_1^{\mathbf{B}} \dots c_n^{\mathbf{B}} u^{(o,o)} v^{(o,o)} x_{k-1}^o \dots x_0^o.M \mid M \in \text{Op}(n, k)\}$$

writing $\tau(n, k)$ as a shorthand for the type $(\mathbf{B}^n, (o, o), (o, o), \overbrace{o, \dots, o}^{k \text{ times}}, o)$. We generalized the notion of representability to terms of type $\tau(n, k)$ as follows:

Definition 3.3.1 (Function pair representation). A closed term $T \in \text{Cl}(\tau(n, k))$ *represents the pair of functions* (f, p) where $f : (\Sigma^*)^n \rightarrow \Sigma^*$ and $p : (\Sigma^*)^n \rightarrow \{0, \dots, k-1\}$ if for all $w_1, \dots, w_n \in \Sigma^*$ and for every $i \in \{0, \dots, k-1\}$ we have:

$$T \underline{w_1} \dots \underline{w_n} =_{\beta\eta} \lambda uvx_{k-1} \dots x_0. \underline{f(w_1, \dots, w_n)} uvx_{|p(w_1, \dots, w_n)|}.$$

By extension we will say that an *open* term M from $\text{Op}(n, k)$ represents the pair (f, p) just if $M[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n] =_{\beta\eta} \underline{f(w_1, \dots, w_n)} uvx_{|p(w_1, \dots, w_n)|}$.

We will call **safe pair** any pair of functions of the form $(w, c(n, i))$ where $0 \leq i \leq k-1$ and w is an n -ary function from $\lambda^{\text{safe}}\text{def}$.

Theorem 3.3.6 (Characterization of the representable pairs). *The function pairs representable in the safe lambda calculus are precisely the safe pairs.*

Proof. (Soundness). Take a pair $(w, c(n, i))$ where $0 \leq i \leq k-1$ and w is an n -ary function from $\lambda^{\text{safe}}\text{def}$. As observed earlier, all the functions from $\lambda^{\text{safe}}\text{def}$ are representable in the safe lambda calculus: let \underline{w} be the representative of w . The pair $(w, c(n, i))$ is then represented by the term $\lambda c_1 \dots c_n uvx_{k-1} \dots x_0. \underline{w} c_1 \dots c_n uvx_i$.

(Completeness) It suffices to consider safe β - η -long normal terms from $\text{Op}(n, k)$ only. The result then immediately follows for any safe term in $\text{Cl}(\tau(n, k))$. The subset of $\text{Op}(n, k)$ constituted of β - η -long normal terms is generated by the following grammar [Zai87]:

$$\begin{array}{lll} (\alpha_i^k) & R^k & \rightarrow x_i \\ (\beta^k) & & | uR^k \\ (\gamma^k) & & | vR^k \\ (\delta_j^k) & & | c_j \overbrace{(\lambda z^k. R^{k+1}[z^k, x_0, \dots, x_{k-1}/x_0, x_1, \dots, x_k])}^{Q^k(R^{k+1})} \\ & & (\lambda z^k. R^{k+1}[z^k, x_0, \dots, x_{k-1}/x_0, x_1, \dots, x_k]) \\ & & R^k \end{array}$$

for $k \geq 1$, $0 \leq i < k$, $0 \leq j \leq n$. The notation $M[\dots/\dots]$ denotes the usual simultaneous substitution. The name of each rule is given in parenthesis. The non-terminals are R^k for $k \geq 1$ and the set of terminals is $\{z^k, \lambda z^k \mid k \geq 1\} \cup \{x_i \mid i \geq 0\} \cup \{c_1, \dots, c_n, u, v\}$.

We identify a rule name with the right-hand side of the corresponding rule, thus α_i^k belongs to $\text{Op}(n, k)$, β^k and γ^k are functions from $\text{Op}(n, k)$ to $\text{Op}(n, k)$, and δ_j^k is a function from $\text{Op}(n, k+1) \times \text{Op}(n, k+1) \times \text{Op}(n, k)$ to $\text{Op}(n, k)$.

We now want to characterize the subset consisted of all *safe* terms generated by this grammar. The term α_i^k is always safe, $\beta^k(M)$ and $\gamma^k(M)$ are safe if and only if M is, and $\delta_j^k(F, G, H)$ is safe if and only if $Q^k(F)$, $Q^k(G)$ and H are safe. We observe that the free variables of $Q^k(F)$ all belong to $\{c_1, \dots, c_n, u, v, x_0, \dots, x_k\}$. All these variables have order greater than $\text{ord } z$ except the x_i s which have same order as z . Hence since the x_i s are not abstracted together with z we have that $Q^k(F)$ is safe if and only if F is safe and the variables $x_0 \dots x_k$ do not appear free in $F[z^k, x_0, \dots, x_{k-1}/x_0, x_1, \dots, x_k]$, which is the same as saying that the variables $x_1 \dots x_k$ do not appear free in F . Similarly, $Q^k(G)$ is safe if and only if G is safe and variables $x_1 \dots x_k$ do not appear free in G .

We therefore need to identify the subclass of terms generated by the non-terminal R^k which are safe and which do not have free occurrences of variables in $\{x_1 \dots x_{k-1}\}$. By applying this requirement to the rules of the previous grammar we obtain the following specialized grammar characterizing the desired subclass:

$$\begin{array}{lll} (\bar{\alpha}_0^k) & \bar{R}^k & \rightarrow x_0 \\ (\bar{\beta}^k) & & | u\bar{R}^k \end{array}$$

$$\begin{array}{ll}
(\bar{\gamma}^k) & | v \bar{R}^k \\
(\bar{\delta}_j^k) & | c_j (\lambda z^k. \bar{R}^{k+1}[z^k/x_0]) (\lambda z^k. \bar{R}^{k+1}[z^k/x_0]) \bar{R}^k .
\end{array}$$

For any term M , $Q^k(M)$ is safe if and only if M can be generated from the non-terminal \bar{R}^k . Thus the subset of $\text{Cl}(\tau(n, k))$ consisting of safe beta-normal terms is given by the grammar:

$$\begin{array}{ll}
(\tilde{\pi}^k) \quad \tilde{S} & \rightarrow \lambda c_1 \dots c_n u v x_{k-1} \dots x_0. \tilde{R}^k \\
(\tilde{\alpha}_i^k) \quad \tilde{R}^k & \rightarrow x_i \\
(\tilde{\beta}^k) & | u \tilde{R}^k \\
(\tilde{\gamma}^k) & | v \tilde{R}^k \\
(\tilde{\delta}_j^k) & | c_j (\lambda z^k. \tilde{R}^{k+1}[z^k/x_0]) (\lambda z^k. \tilde{R}^{k+1}[z^k/x_0]) \tilde{R}^k .
\end{array}$$

Thus to conclude the proof, it suffices to show that every term that can be generated by this grammar starting with the non-terminal \tilde{S} represents a safe pair.

We proceed by induction and show that the non-terminal \bar{R}^k generates terms representing pairs of the form $(w, c(n, 0))$ while non-terminals \tilde{S} and \tilde{R}^k generate terms representing pairs of the form $(w, c(n, i))$ for $0 \leq i < k$ and $w \in \lambda^{safe} \text{def}$.

Base case: The term $\bar{\alpha}_0^k$ represents the safe pair $(c(n, 0), c(n, 0))$ while $\tilde{\alpha}_i^k$ represents the safe pair $(c(n, 0), c(n, i))$. *Step case:* Suppose $T \in \text{Op}(n, k)$ represents a pair (w, p) . Then $\bar{\alpha}^k(T)$ and $\tilde{\alpha}^k(T)$ represent the pair $(\text{app}(a, w), p)$, $\bar{\beta}^k(T)$ and $\tilde{\beta}^k(T)$ represent the pair $(\text{app}(b, w), p)$, and $\bar{\pi}^k(T) \in \text{Cl}(\tau(n, k))$ represents the pair (w, p) . Now suppose that E, F and G represent the pairs $(w_e, c(n, 0))$, $(w_f, c(n, 0))$ and $(w_g, c(n, i))$ respectively. Then we have:

$$\begin{aligned}
& \tilde{\delta}_j^k(E, F, G)[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n] \\
&= \underline{w_j} (\lambda z^k. E[z^k/x_0])[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n] \\
&\quad (\lambda z^k. F[z^k/x_0])[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n] \\
&\quad G[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n] \\
&=_{\beta\eta} \underline{w_j} (\lambda z^k. E[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n][z^k/x_0]) \\
&\quad (\lambda z^k. F[\underline{w_1} \dots \underline{w_n}/c_1 \dots c_n][z^k/x_0]) \\
&\quad (\underline{w_g(w_1 \dots w_n)} u v x_i) \quad \text{G represents } (h, c(n, i)) \\
&=_{\beta\eta} \underline{w_j} (\lambda z^k. (\underline{w_e(w_1 \dots w_n)} u v x_0)[z^k/x_0]) \quad \text{E represents } (f, c(n, 0)) \\
&\quad (\lambda z^k. (\underline{w_f(w_1 \dots w_n)} u v x_0)[z^k/x_0]) \quad \text{F represents } (g, c(n, 0)) \\
&\quad (\underline{w_g(w_1 \dots w_n)} u v x_i) \\
&=_{\beta\eta} \underline{w_j} (\lambda z^k. \underline{w_e(w_1 \dots w_n)} u v z^k) \\
&\quad (\lambda z^k. \underline{w_f(w_1 \dots w_n)} u v z^k) \\
&\quad (\underline{w_g(w_1 \dots w_n)} u v x_i) \\
&=_{\eta} \underline{w_j} (\underline{w_e(w_1 \dots w_n)} u v) (\underline{w_f(w_1 \dots w_n)} u v) (\underline{w_g(w_1 \dots w_n)} u v x_i) \\
&=_{\beta\eta} \underline{w_j} u v x_i
\end{aligned}$$

where the word-function w is defined as

$$w : w_1, \dots, w_n \mapsto \text{app}(\text{sub}(w_j, w_e(w_1, \dots, w_n), w_f(w_1, \dots, w_n)), w_g(w_1, \dots, w_n)) .$$

Hence $\tilde{\delta}_j^k(E, F, G)$ represents the pair $(w, c(n, i))$.

The same argument shows that if E, F and G all represent safe pairs then so does $\bar{\delta}_j^k(E, F, G)$. \square

By instantiating Theorem 3.3.6 with terms of types $\tau(n, 1) = I^n \rightarrow I$ we obtain that every closed safe term of this type represents some n -ary function from λ^{safe} def. This concludes the proof of the characterization Theorem 3.3.5.

3.4 Typing problems

In this section we consider the problems of type checking, typability and type inhabitation as defined in Sec. 2.1 but recast in the safe lambda calculus:

- TYPE CHECKING: Given a term M , context Γ and type A , do we have $\Gamma \vdash_s M : A$?
- TYPABILITY: Given a term M and context Γ , is there a type A such that $\Gamma \vdash_s M : A$?
- INHABITATION: Given a type A , is there a term M such that $\vdash_s M : A$?

We will restrict our attention to the Church-like safe lambda calculus. The results presented here straightforwardly extend to the Curry version.

3.4.1 Relating derivations from $\Lambda_{\rightarrow}^{Cu}$ and safe $\Lambda_{\rightarrow}^{Cu}$

In this section we compare derivations obtained in the simply typed lambda calculus with those obtained in the safe lambda calculus. In order to ease our comparison, we use an alternative presentation of the simply typed lambda calculus given in Table 3.4. The difference with the typing-system from Def. 2.1.9 is the presence of the weakening rule and the ability to perform simultaneous application and abstraction at once. The two presentations are clearly equivalent:

$$\begin{array}{c}
 \frac{}{x : A \vdash_{Cu} x : A} \quad \frac{\Gamma \vdash_{Cu} M : A}{\Delta \vdash_{Cu} M : A} \quad \Gamma \subset \Delta \\
 \\
 \frac{\Gamma \vdash_{Cu} M : (A_1, \dots, A_n, B) \quad \Gamma \vdash_{Cu} N_1 : A_1 \quad \dots \quad \Gamma \vdash_{Cu} N_n : A_n}{\Gamma \vdash_{Cu} M N_1 \dots N_n : B} \\
 \\
 \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \vdash_{Cu} M : B}{\Gamma \vdash_{Cu} \lambda x_1 \dots x_n. M : (A_1, \dots, A_n, B)}
 \end{array}$$

Table 3.4: Alternative definition of the Curry-style lambda calculus

$\Gamma \vdash_{Cu} M : T$ is derivable in this system iff it is derivable with the rules of Def. 2.1.9.

CONVENTION 3.4.1 In order to make our derivations canonical, we adopt the following convention:

- a derivation cannot contain two consecutive applications of the weakening rule;
- when using the weakening rule, the context Δ is chosen as small as possible so that for every judgement $\Gamma \vdash_{Cu} M : A$ appearing in the derivation that is not deduced from the weakening rule we have $FV(M) = \text{dom}(\Gamma)$.

We are interested in those derivations verifying the following property: a deduction Δ of $\Gamma \vdash_{Cu} M : T$ is **compact** if the set of terms appearing in the nodes of the deduction tree Δ is precisely $\text{sub}(M)$. In other words, in a compact deduction, each use of the application and abstraction rule in the deduction is as “large” as possible so that each path in the deduction tree is constituted of an axiom followed by an alternation of application/abstraction rules. Compact derivations are sufficient: if there is derivation in $\Lambda_{\rightarrow}^{Cu}$ then there is a compact derivation with the same conclusion. We will write $\text{Der}_{cu}(\Gamma, M, T)$ for the set of compact derivations of $\Gamma \vdash_{Cu} M : T$.

Similarly, we define the notion of compact derivation in the safe lambda calculus. It is easy to check that, despite the side-conditions imposed by the abstraction rule, the compact deductions

are sufficient. We write $\text{Der}_s(\Gamma, M, T)$ for the set of compact deductions of $\Gamma \vdash_s M : T$ in safe $\Lambda_{\rightarrow}^{\text{Cu}}$.

We say that a deduction $\Delta \in \text{Der}_{cu}(\Gamma, M, T)$ is *safe* if $\text{ord } \Gamma \geq \text{ord } T$ and for all term-in-context $\Gamma' \vdash_{\text{st}} M : T'$ deduced in Δ from the abstraction rule we have: $\text{ord } \Gamma' \geq \text{ord } T'$.

For any deduction tree Δ in $\text{Der}_s(\Gamma, M, T)$ we write $\epsilon(\Delta)$ to denote the deduction tree obtained by replacing judgements $\Gamma \vdash_s M : T$ by $\Gamma \vdash_{\text{Cu}} M : T$ and rules of the safe lambda calculus by their counterpart in the simply type lambda calculus (identifying (app) and (app_{as})).

Lemma 3.4.1 (Relating derivations from $\Lambda_{\rightarrow}^{\text{Cu}}$ and safe $\Lambda_{\rightarrow}^{\text{Cu}}$).

- (i) $\Delta \in \text{Der}_s(\Gamma, M, T) \implies \epsilon(\Delta) \in \text{Der}_{cu}(\Gamma, M, T) \wedge \epsilon(\Delta) \text{ is safe,}$
- (ii) $\Delta' \in \text{Der}_{cu}(\Gamma, M, T) \wedge \Delta' \text{ is safe.} \implies \exists \Delta \in \text{Der}_s(\Gamma, M, T) : \Delta' = \epsilon(\Delta).$

Proof. This follows immediately from the definition of safe $\Lambda_{\rightarrow}^{\text{Cu}}$. □

3.4.2 Type checking and typability

By the PT Theorem 2.1.4, if a term is typable then it has a computable principal derivation: every other derivation is an instance of that derivation. The same result holds for compact derivations:

Lemma 3.4.2 (Principal compact derivation). *If M is typable in $\Lambda_{\rightarrow}^{\text{Cu}}$ then it has a compact principal derivation Δ (i.e., any derivation $\Delta' \in \text{Der}_{cu}(\Gamma, M, T)$ is an instance of Δ) that is computable from M .*

Proof. This follows immediately from Theorem 2.1.4: compact derivations are just “reorganized” derivations: for any standard derivation there exists a corresponding compact derivation containing the same typing assumptions. The compact principal derivation can be obtained from the principal derivation by performing the same “reorganization”. □

Proposition 3.4.1. TYPE CHECKING in safe $\Lambda_{\rightarrow}^{\text{Cu}}$ is decidable.

Proof. Let $M \in \Lambda$, $T \in \mathbb{T}$ and Γ be a typing-context. We have $\Gamma \vdash_s M : T$ iff $\text{Der}_s(\Gamma, M, T) \neq \emptyset$. By Lemma 3.4.1, there is a derivation in $\text{Der}_s(\Gamma, M, T)$ if and only if there is a safe derivation in $\text{Der}_{cu}(\Gamma, M, T)$. We already know that the TYPE CHECKING problem in $\Lambda_{\rightarrow}^{\text{Cu}}$ (“Is $\text{Der}_{cu}(\Gamma, M, T)$ empty?”) is decidable. If $\text{Der}_{cu}(\Gamma, M, T)$ is empty then we can answer ‘No’ to the type-checking problem. Otherwise by the previous Lemma, we can compute a compact principal derivation Δ_p of $\Gamma \vdash_s M : T$ and we know that there exists a safe derivation iff there exists a type-substitution s for Δ_p such that (i) $s(\Delta_p)$ is safe; (ii) the conclusion of $s(\Delta_p)$ is $\Gamma \vdash_s M : T$.

The latter property can be decided by unifying the types appearing in the conclusion of Δ_p with Γ and T . The former property turns out to be also decidable. Indeed, the deduction Δ_p contains finitely many atoms $a_1 \dots a_n \in \mathbb{A}$, $n \geq 1$. Therefore the safety of $s(\Delta_p)$ can be expressed in terms of a system of inequations over the order of the atoms occurring in Δ_p . This system can be reexpressed into a system of inequations \mathcal{S} of the form $x_i > x_j$ for $i, j \in \{1, \dots, q\}$ and variables $x_1, \dots, x_q \in \mathbb{Z}$ and such that for every atom a_k , $\text{ord } a_k = x_{i_k}$ for some $i_k \in \{1, \dots, q\}$.

A substitution s verifying the required property exists if and only if \mathcal{S} has a solution, and if the solution to \mathcal{S} is (x_1, \dots, x_q) then we just need to take the substitution $s = [(x_{k_1})_o/a_1, \dots, (x_{k_n})_o/a_n]$ for some fresh atom $o \in \mathbb{A}$. (Observe that if (x_1, \dots, x_q) is a solution then so is $(x_1 + k, \dots, x_q + k)$ for $k \geq 0$, therefore the x_i s can all be assumed to be positive.) The system \mathcal{S} can then be solved using a topological sorting algorithm [Knu00]. □

Proposition 3.4.2. TYPABILITY in safe $\Lambda_{\rightarrow}^{\text{Cu}}$ is decidable.

Proof. The proof is the same as for TYPE CHECKING except that only condition (i) needs to be decided. □

3.4.3 The type inhabitation problem

Statman showed that the problem of deciding whether a type *defined over an infinite number of ground atoms* is inhabited (or equivalently of deciding validity of an intuitionistic implicative formula) is PSPACE-complete [Sta79a]. In the safe lambda calculus, no complexity is known. In fact it is not even clear whether the problem is decidable:

Proposition 3.4.3. *INHABITATION in safe Λ_{\perp} is (at least) semi-decidable: given a simple type, there is an algorithm that prints a safe inhabitant if there is one but may not terminate if there is not.*

Proof. Inhabitants are enumerated using Ben-Yelles’s counting algorithm [Hin97]. Each inhabitant can be tested for typability in safe Λ_{\perp} by Proposition 3.4.2. \square

It is well known that the simply typed lambda calculus corresponds to intuitionistic implicative logic via the Curry-Howard isomorphism. The theorems of the logic correspond to inhabited types; further every inhabitant of a type represents a proof of the corresponding formula. Similarly, we can consider the fragment of intuitionistic implicative logic that corresponds to the safe lambda calculus under the Curry-Howard isomorphism. We call it the *safe fragment of intuitionistic implicative logic*.

An obvious question is to compare the reasoning power of these two logics, in other words, to determine which types are inhabited in the lambda calculus but not in the safe lambda calculus.¹ Since safety is preserved by β -reduction, it is enough to look at inhabitants that are in β -normal form, also called *normal inhabitants*. We say that a type is **unsafe** if it is inhabited and every inhabitant is unsafe. At order 2, all closed normal terms are safe therefore there is no unsafe type at this order. The following proposition further shows that types generated from a single atom o are all not unsafe:

Proposition 3.4.4. *Every type generated from one atom o that is inhabited in the lambda calculus is also inhabited by a safe lambda term.*

Proof. One can transform any unsafe normal inhabitant M into a safe one of the same type as follows: Compute the eta-long beta-normal form of M . Let x be an occurrence of a ground-type variable in a subterm of the form $\lambda\bar{x}.C[x]$ where $\lambda\bar{x}$ is the binder of x and for some context $C[-]$ different from the identity ($C[-] \equiv -$). Since the term is considered is beta-normal and because its type is built out of a unique atom o , x is necessarily of type o . We then replace the subterm $\lambda\bar{x}.C[x]$ by $\lambda\bar{x}.x$ in M . This transformation is sound because $C[x]$ and x both have type o . We repeat this procedure until the term stabilizes. This algorithm clearly terminates since the size of the term decreases strictly after each step. The final term obtained is safe and of the same type as M . \square

The previous argument crucially uses the fact that the type is generated from a single atom. It cannot be repeated for types generated from multiple atoms. In fact there are order-3 types with only 2 atoms that are inhabited by simply typed terms but not by safe terms as example (i) below shows.

Example 3.4.1. Let a, b and c be three distinct atoms.

- (i) Take the order-3 type $((b, a), b), ((a, b), a), a$. Its normal inhabitants are given (up to α -conversion) by the following family of terms which are all unsafe:

$$\begin{aligned} &\lambda f g. g(\lambda x_1. f(\lambda y_1. x_1)) \\ &\lambda f g. g(\lambda x_1. f(\lambda y_1. g(\lambda x_2. y_1))) \\ &\lambda f g. g(\lambda x_1. f(\lambda y_1. g(\lambda x_2. f(\lambda y_2. x_i)))) \quad \text{where } i = 1, 2 \\ &\lambda f g. g(\lambda x_1. f(\lambda y_1. g(\lambda x_2. f(\lambda y_2. g(\lambda x_3. y_i)))) \quad \text{where } i = 1, 2 \\ &\dots \end{aligned}$$

¹This problem was raised by Ugo dal Lago.

- (ii) The order-3 type $((a, c), b), ((c, b), a), a$ has for only normal inhabitant the unsafe term $\lambda f g. g(\lambda x. f(\lambda y. c))$.
- (iii) For any $i, j, k \in \mathbb{N}$, let $\sigma(i, j, k)$ denote the type

$$\sigma(i, j, k) \equiv (i_a \rightarrow j_b) \rightarrow (j_b \rightarrow k_c) \rightarrow i_a \rightarrow k_c$$

where $(n+1)_a$ denotes the type $(\dots((a \rightarrow a) \rightarrow a) \dots) \rightarrow a$ containing n occurrence of a as defined in Sec. 2.1.5. This type is inhabited by the “function composition term”:

$$\lambda x y z w. y(xz)$$

which is safe if and only if $i \geq j$. There exists values for i, j, k such that $i < j$ and $\sigma(i, j, k)$ is safely inhabited. For instance $\sigma(1, 3, 4)$ is inhabited by the safe term

$$\lambda x^{1_a \rightarrow 3_b} y^{3_b \rightarrow 4_c} z^{1_c} w^{3_c}. y(x(\lambda u^a. u)) \text{ .}$$

The order-4 type $\sigma(0, 2, 0)$, however, is unsafe: its only normal inhabitant is the unsafe term $\lambda x y z w. y(xz)$.

(The first two examples are due to Luke Ong.)

3.5 Extensions


We now consider extensions of the safe simply type lambda calculus.

3.5.1 PCF

We define **safe PCF** as the applied safe lambda calculus with types defined over a single atomic type of natural numbers and extended with the basic operators of PCF: additions, substraction, conditional branching and recursion). Equivalently, it is the restriction of PCF where the application and abstraction rules are constrained similarly as in the safe lambda calculus. The rules are given in Table 3.5. The circled rules are those that differ from their PCF counterpart.

We extend the notion of *almost safety* (Sec. 3.1.4) to PCF: a PCF term is **almost safe** if it can be written $\lambda x_1 \dots x_n. N_0 \dots N_p$ for some $n, p \geq 0$ where N_i is safe for all $0 \leq i \leq p$.

Example 3.5.1. The addition function and equality test defined in Sec. 2.1.9 are typable in safe PCF.

The Substitution  Lemma and No-variable-capture Lemma shown for the safe lambda calculus naturally extends to safe PCF. The small-step semantics of safe PCF is obtained from the one of PCF by replacing β -reduction by safe β -reduction (Def. 3.1.5) in the definition of the relation \rightarrow . The Subject Reduction Lemma from the safe lambda calculus implies that the relation \rightarrow preserves safety: Suppose that $M \rightarrow N$, then $\Gamma \vdash_s M : T$ implies $\Gamma \vdash_s N : T$. It also preserves almost-safety. Finally, a term is safe if and only if its eta-long normal form is safe.

Remark concerning recursion

There are many ways to introduce recursion in the syntax of a programming language. In the presentation of PCF given in Sec. 2.1.9, recursion is introduced by mean of a set of constants Y_A , A ranging over PCF types, incarnating the Y -combinator of the lambda calculus. Its syntax is given by the rule (rec) of Table 3.5. For instance, the addition function can be represented by the PCF term:

$$\text{PLUS} \equiv Y(\lambda p x y. \text{cond } x y (p (\text{pred } x) (\text{succ } y))) \text{ .}$$

Equivalently, we can introduce recursion using the *least upper bound abstractor* ‘ μ ’ given by the formation rule

$$(\mu) \frac{\Gamma, f : A \vdash M : A}{\Gamma \vdash \mu f^A. M : A}$$

Functional part

$$\begin{array}{l}
(\text{var}) \frac{}{\Gamma \vdash_s x : A} \quad x : A \in \Gamma \quad (\text{wk}) \frac{\Gamma \vdash_s M : A}{\Delta \vdash_s M : A} \quad \Gamma \subset \Delta \quad (\delta) \frac{\Gamma \vdash_s M : A}{\Gamma \Vdash_{\text{app}} M : A} \\
\\
(\text{app}_{\text{as}}) \frac{\Gamma \vdash_s M : (A_1, \dots, A_n, B) \quad \Gamma \vdash_s N_1 : A_1 \quad \dots \quad \Gamma \vdash_s N_n : A_n}{\Gamma \Vdash_{\text{app}} M N_1 \dots N_n : B} \\
\\
(\text{app}) \frac{\Gamma \vdash_s M : (A_1, \dots, A_n, B) \quad \Gamma \vdash_s N_1 : A_1 \quad \dots \quad \Gamma \vdash_s N_n : A_n}{\Gamma \vdash_s M N_1 \dots N_n : B} \quad \text{ord } B \leq \text{ord } \Gamma \\
\\
(\text{abs}) \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \Vdash_{\text{app}} M : B}{\Gamma \vdash_s \lambda x_1^{A_1} \dots x_n^{A_n}. M : (A_1, \dots, A_n, B)} \quad \text{ord } (A_1, \dots, A_n, B) \leq \text{ord } \Gamma
\end{array}$$

Arithmetic and recursion

$$\begin{array}{l}
(\text{const}) \frac{}{\vdash_s n : \text{exp}} \quad (\text{succ}) \frac{\Gamma \vdash_s M : \text{exp}}{\Gamma \vdash_s \text{succ } M : \text{exp}} \quad (\text{pred}) \frac{\Gamma \vdash_s M : \text{exp}}{\Gamma \vdash_s \text{pred } M : \text{exp}} \\
\\
(\text{cond}) \frac{\Gamma \vdash_s M : \text{exp} \quad \Gamma \vdash_s N_1 : \text{exp} \quad \Gamma \vdash_s N_2 : \text{exp}}{\Gamma \vdash_s \text{cond } M N_1 N_2} \quad (\text{rec}) \frac{\Gamma \vdash_s M : A \rightarrow A}{\Gamma \vdash_s Y_A M : A}
\end{array}$$

Table 3.5: Formation rules for Safe PCF.

where the semantics of μ is given by the rule: $\mu f^A.M \rightarrow M[(\mu f^A.M)/f]$. For instance, the addition function can be defined using the μ -construct as:

$$\text{PLUS} \equiv \mu p^{(\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}}. \lambda x^{\text{exp}} y^{\text{exp}}. \text{cond } x y (p (\text{pred } x) (\text{succ } y)) .$$

Clearly in the context of PCF, these two definitions are interchangeable: $\mu f^A.M$ is equivalent to $Y_A(\lambda f^A.M)$, and $Y_A F$ is eta-equivalent to $Y_A(\lambda f^A.Ff)$ for some fresh variable f , which is equivalent to $\mu f^A.Ff$.

In the context of safe PCF, however, the distinction is important. Indeed, let *safe μ -PCF* denote the calculus obtain by replacing the rule (rec) by (μ) in Table 3.5. It turns out that safe PCF is strictly contained in safe μ -PCF. Indeed, compare the two ways of defining a recursive term:

$$\begin{array}{l}
(\text{abs}) \frac{\Gamma, f : A \vdash_s M : A}{\Gamma \vdash_s \lambda f^A.M : A \rightarrow A} \\
(\text{rec}) \frac{}{\Gamma \vdash_s Y_A(\lambda f^A.M)} \\
\\
(\mu) \frac{\Gamma, f : A \vdash_s M : A}{\Gamma \vdash_s \mu f^A.M : A}
\end{array}$$

Both derivations start with the premise $\Gamma, f : A \vdash_s M : A$ which implies that $\text{ord } \Gamma \geq \text{ord } A$. In the left derivation, before applying the Y combinator, we need first to abstract the variable f . This is done using the abstraction rule whose side-conditions gives $\text{ord } \Gamma > \text{ord } A$. The right derivation, however, only imposes the condition $\text{ord } \Gamma \geq \text{ord } A$.

In fact, safe μ -PCF does not deserve its name because the No-variable-capture lemma does not hold anymore in this language! For instance for any types A and B with $\text{ord } A \geq \text{ord } B$, the term

$$\lambda f^{A \rightarrow B} a^A. (\lambda x^B. (\mu f^B.x))(fa)$$

is a safe μ -PCF term that β -reduces to

$$\lambda f^{A \rightarrow B} a^A. (\mu f^B.x)[fa/x] .$$

At this point it would not be sound to push the substitution under the μ without first renaming the variables afresh as it would cause the variable f to be captured by μf . Hence the definition that we really want for safe PCF is the one based on the Y-combinator.

Observe that if we were able to distinguish variables that are bound by λ from those bound by μ —for instance by tagging their occurrences appropriately—then the clash of variable names would be tolerable in this particular example since the two clashing occurrences of f are bound by a different kind of binder. Unfortunately, this argument cannot be generalized: there are safe μ -PCF terms that, when reduced using capture-permitting substitution, cause clashes between λ -bound variables. Take for instance:

$$M \equiv \lambda g^3 h^3 x^1. g(\mu F^3. N(F, g, h, x))$$

$$N(F, g, h, x) \equiv x(h(\lambda x^1. F(\lambda z^0. z)))$$

where 0 denotes the type o and $n + 1$ denotes $n \rightarrow o$, for $n \in \mathbb{N}$. The safe μ -PCF term M reduces to:

$$\lambda g^3 h^3 x^1. g(x(h(\lambda \underline{x}^1. F(\lambda z^0. z))))[N(F, g, h, \underline{x})/F] ,$$

and performing this substitution *capture-permitting* would cause a clash between the two underlined variables.

Another reason why safe μ -PCF is not an interesting language is that the game-semantic characterization of safe PCF that we will establish in Chapter 6 does not hold in safe μ -PCF.

3.5.1.1 Expressivity

In the lambda calculus, the safety condition significantly limits the expressivity of the language. The conditional over Church numerals, for instance, is not definable in the safe lambda calculus. In safe PCF since the arithmetic constructs are built in the language, the conditional operator comes for free. In fact, despite the strong syntactic constraint imposed on the language, the presence of recursion gives safe PCF the computational power of a full-fledged Turing complete language.

We first show that termination of safe PCF term is not decidable by a reduction from the QUEUE-HALTING problem.

The Queue programming system We fix a finite alphabet $\Sigma = \{a_1, \dots, a_p\}$. A QUEUE program is a finite sequence of instructions that manipulate a FIFO (First In First Out) queue data-structure. A program P is a sequence of n instructions for some $n \in \mathbb{N}$. For $1 \leq i \leq n$ we write $P.i$ to denote the i^{th} instruction of P . There are four kinds of instruction: halting, enqueueing, dequeuing and branching. The set of instructions is given by:

$$\mathcal{I} = \{\text{halt}\} \cup \{\text{enqueue } a \mid a \in \Sigma\} \cup \{\text{dequeue}\} \cup \{\text{goto } l \text{ if first} = a \mid l \in 1..n, a \in \Sigma\}$$

The operational semantics is described using a set of states $\{\text{halted}\} \cup \{1, \dots, n\} \times \Sigma^*$. The end-of-program state **halted** corresponds to the final state reach when the program terminates. A state of the form $(i, x) \in \{1, \dots, n\} \times \Sigma^*$ indicates that the queue's content is given by the sequence x and that the next instruction to be executed by the machine is $P.i$. The empty queue is represented by the empty sequence ϵ , and for any sequence $x \in \Sigma^*$, the first element of x corresponds to the element that has been *first* enqueued (*i.e.*, the queue is fed at the right-end side and consumed at the left-end side). The operational semantics is defined by the following rules:

$$\begin{aligned} (i, x) \text{ with } P.i = \text{halt} &\rightarrow \text{halted} \\ (i, x) \text{ with } P.i = \text{enqueue } a &\rightarrow (i + 1, x \cdot a) \\ (i, \epsilon) \text{ with } P.i = \text{dequeue} &\rightarrow \text{halted} \\ (i, a \cdot x) \text{ with } P.i = \text{dequeue} &\rightarrow (i + 1, x) \\ (i, \epsilon) \text{ with } P.i = \text{goto } l \text{ if first} = a &\rightarrow (i + 1, \epsilon) \\ (i, b \cdot x) \text{ with } a \neq b \text{ and } P.i = \text{goto } l \text{ if first} = a &\rightarrow (i + 1, b \cdot x) \end{aligned}$$

$$(i, a \cdot x) \text{ with } P.i = \text{goto } l \text{ if first} = a \rightarrow (l, a \cdot x)$$

We write \rightarrow^* to denote the reflexive transitive closure of \rightarrow .

The QUEUE-HALTING problem (“Given a QUEUE program, will it halt eventually?”) is undecidable. This is because Post’s Tag Systems, which are Turing complete [CM64], can be simulated [Min67] in QUEUE.

Encoding Queue-Halting in safe PCF Given a QUEUE program P with n instructions, we construct a safe PCF term $\vdash_s M_P : \mathbf{exp}$ that simulates P in the sense that $P \Downarrow$ if and only if $M_P \rightarrow^*$ halted.

We fix a distinguished element \perp denoting the end of the queue. Let $\Sigma^\perp = \Sigma \cup \{\perp\}$. The queue content $s \in \Sigma^*$ is identified with the infinite sequence $s\perp^\omega \in \Sigma^\omega$. We assume that an injective encoding function $\Sigma^\perp \rightarrow \mathbb{N}$ is given. For instance take $\overline{1} = 0$ and $\overline{a_k} = k$ for $1 \leq k \leq p$.

We say that a PCF term M computes the queue content s if and only if $M k \Downarrow \overline{s_k}$ for all $k \in \mathbb{N}$. For any queue-content $s \in \Sigma^*$ we define the safe PCF term

$$\vdash_s \overline{s} \equiv \lambda i^{\mathbf{exp}}. \text{match } i \text{ with } 0 \rightarrow \overline{s_0} \mid \dots \mid n \rightarrow \overline{s_{|s|-1}} \mid _ \rightarrow \overline{1} : \mathbf{exp} \rightarrow \mathbf{exp}$$

which clearly computes s . The length $|s|$ of the queue can then be computed by the term

$$\begin{aligned} \vdash_s \text{LENGTH} &\equiv Y(\lambda f^{\mathbf{exp} \rightarrow (\mathbf{exp} \rightarrow \mathbf{exp}) \rightarrow \mathbf{exp}} k^{\mathbf{exp}} x^{\mathbf{exp} \rightarrow \mathbf{exp}} . \\ &\quad \text{if } x k = \overline{1} \text{ then } k \text{ else } f(k+1)x) 0 : (\mathbf{exp} \rightarrow \mathbf{exp}) \rightarrow \mathbf{exp} \end{aligned}$$

verifying $\text{LENGTH } \overline{s} \Downarrow |s|$ for all $s \in \Sigma^*$.

Each instruction c of \mathcal{I} is encoded into a unique natural number \overline{c} by some function $\mathcal{I} \rightarrow \mathbb{N}$. For instance we can use the following injection: for $1 \leq i \leq p, 1 \leq l \leq n$,

$c \in \mathcal{I}$	halt	dequeue	enqueue a_i	goto l if first = a_i
$\overline{c} \in \mathbb{N}$	0	1	$1+i$	$1+p+n.l+i$

The QUEUE program P is then represented by the safe PCF term:

$$\vdash_s \overline{P} \equiv \lambda i^{\mathbf{exp}}. \text{match } i \text{ with } 0 \rightarrow \overline{P.0} \mid \dots \mid n \rightarrow \overline{P.n} \mid _ \rightarrow \overline{\text{halt}} : \mathbf{exp} \rightarrow \mathbf{exp}$$

so that for all $i \in \mathbb{N}$, $\overline{P}i$ evaluates to the encoding of the i^{th} instruction of P . The following term defines an interpreter for QUEUE-programs given in “compiled” form \overline{P} :

$$\begin{aligned} \vdash_s \text{SIM} &\equiv Y(\lambda f^{(\mathbf{exp}, (\mathbf{exp}, \mathbf{exp}), \mathbf{exp})} i^{\mathbf{exp}} x^{(\mathbf{exp}, \mathbf{exp})} . \\ &\quad \text{match } \overline{P}i \text{ with} \\ &\quad \quad \overline{\text{halt}} \rightarrow 0 \\ &\quad \quad \overline{\text{dequeue}} \rightarrow f(i+1)(\lambda j^{\mathbf{exp}}. x(j+1)) \\ &\quad \quad \overline{\text{enqueue } a_1} \rightarrow f(i+1)(\lambda j^{\mathbf{exp}}. \text{if } j = \text{LENGTH } x \text{ then } \overline{a_1} \text{ else } x j) \\ &\quad \quad \dots \\ &\quad \quad \overline{\text{enqueue } a_p} \rightarrow f(i+1)(\lambda j^{\mathbf{exp}}. \text{if } j = \text{LENGTH } x \text{ then } \overline{a_p} \text{ else } x j) \\ &\quad \quad \overline{\text{goto } l \text{ if first} = a_1} \rightarrow \text{if LENGTH } x = 0 \text{ then } f(i+1)x \\ &\quad \quad \quad \text{else if } \overline{a_1} = x 0 \text{ then } f l x \\ &\quad \quad \quad \text{else } f(i+1)x \\ &\quad \quad \dots \\ &\quad \quad \overline{\text{goto } l \text{ if first} = a_p} \rightarrow \text{if LENGTH } x = 0 \text{ then } f(i+1)x \\ &\quad \quad \quad \text{else if } \overline{a_p} = x 0 \text{ then } f l x \\ &\quad \quad \quad \text{else } f(i+1)x \\ &\quad) 0 \overline{c} : \mathbf{exp} \end{aligned}$$

Clearly the term SIM is safe and simulates the QUEUE program P in the sense that $\text{SIM} \Downarrow$ if and only if $P \rightarrow^*$ halted. Hence

Theorem 3.5.1. *The HALTING problem for (the 2nd order fragment of) safe PCF is undecidable.*

Since the HALTING is reducible to the observational equivalence problem, this also implies that observational equivalence for the 2nd-order fragment of safe PCF (with Y_1 recursion and unbounded base types) is undecidable. This result is in fact not surprising: it is not difficult to see that the partial recursive functions are computable in the order 2 fragment of safe PCF, and hence safe PCF is Turing complete. (This can also be proved by simulating Turing machines in safe PCF using an encoding similar to the one used above.)

The reason why these encodings work is because unsafety only appears at order 3 in PCF, and the 2nd order fragment of PCF is already Turing complete.

Loader has shown [Loa01] that observational equivalence for *Finitary* PCF (the fragment with no recursion and finite base types) is already undecidable at order 5. It is unknown whether this result still holds for Finitary safe PCF.

3.5.2 Idealized Algol

In this section we present two possible approaches to accommodate the safety restriction to a language featuring block-variable constructs such as Idealized Algol. This gives rise to two different versions of “Safe Idealized Algol”. In the first version, all free variables are required to satisfy the safety constraint whereas in the second version, variables declared with a block-allocated construct are not required to satisfy the safety constraint. We then show that the nice properties of the safe lambda-calculus remain in these two extensions of the safe lambda calculus.

3.5.2.1 Strongly Safe IA

The most immediate way to introduce the safety constraint for IA terms consists in adding the typing rules for constants of IA to those of the safe lambda calculus. Equivalently, this means taking the system of rules of IA and replacing the application and abstraction rules by those of the safe lambda calculus. We refer to this language as **strongly safe IA**. The rules are formally given in Table 3.7. The rules circled in the table are those that differ from their IA counterpart.

This language verifies the basic property of the safe lambda calculus: free variables have order greater or equal to the order of the term. It is interesting to note that the typing rules of IA do not need to be modified for this property to hold. In particular, the rule (new) allows one to “abstract” variables without having to verify any side-condition, contrary to the lambda-abstraction rule (abs). Such side-condition is unnecessary because the block-allocation construct produces a term with the same type as the term in the premise of the rule. Therefore the basic property trivially holds.

On the other hand, this ability to “abstract” variables without increasing the order of the term as a downside: the No-variable-capture result—that it is no necessary to rename variables afresh when performing substitution—does not hold anymore, at least in its original formulation. Take for instance the following strongly-safe term-in-context:

$$x : \text{var} \vdash_{ss} (\lambda y^{\text{exp}}. \text{new } x \text{ in } y)(\text{deref } x) \equiv M_1 : \text{exp} .$$

Then we have:

$$M_1 \rightarrow_{\beta} (\text{new } x \text{ in } y) [(\text{deref } x)/y] .$$

Performing the substitution without renaming variables afresh causes the variable x to get captured by the innermost **new** x giving: **new** x in **deref** x . On the other hand the standard substitution gives: **new** z in **deref** x . These two terms are clearly not observationally equivalent. Conclusion: it is *not* “safe” to use capture-permitting substitution on strongly-safe IA terms!

A weaker version of the No-variable-capture lemma can be stated though. We can defined an alternative notion of capture-permitting substitution, called *semi-capture permitting substitution*, that behaves like the usual capture-permitting substitution except that it renames block-allocated variables afresh upon performing substitution. The No-variable-capture lemma for strongly safe IA then becomes: substitution can be safely implemented by semi-capture permitting substitution.

Functional part

$$\begin{array}{l}
(\text{var}) \frac{}{\Gamma \vdash_{ss} x : A} \quad x : A \in \Gamma \quad (\text{wk}) \frac{\Gamma \vdash_{ss} s : A}{\Delta \vdash_{ss} s : A} \quad \Gamma \subset \Delta \quad (\delta) \frac{\Gamma \vdash_{ss} M : A}{\Gamma \Vdash_{\text{app}} M : A} \\
(\text{app}_{\text{as}}) \frac{\Gamma \vdash_{ss} s : (A_1, \dots, A_n, B) \quad \Gamma \vdash_{ss} t_1 : A_1 \quad \dots \quad \Gamma \vdash_{ss} t_n : A_n}{\Gamma \Vdash_{\text{app}} s \, t_1 \dots t_n : B} \\
(\text{app}) \frac{\Gamma \vdash_{ss} s : (A_1, \dots, A_n, B) \quad \Gamma \vdash_{ss} t_1 : A_1 \quad \dots \quad \Gamma \vdash_{ss} t_n : A_n}{\Gamma \vdash_s s \, t_1 \dots t_n : B} \quad \text{ord } B \leq \text{ord } \Gamma \\
(\text{abs}) \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \Vdash_{\text{app}} s : B}{\Gamma \vdash_{ss} \lambda x_1 \dots x_n. s : (A_1, \dots, A_n, B)} \quad \text{ord } (A_1, \dots, A_n, B) \leq \text{ord } \Gamma
\end{array}$$

Arithmetic and recursion

$$\begin{array}{l}
(\text{const}) \frac{}{\vdash_{ss} n : \text{exp}} \quad (\text{succ}) \frac{\Gamma \vdash_{ss} M : \text{exp}}{\Gamma \vdash_{ss} \text{succ } M : \text{exp}} \quad (\text{pred}) \frac{\Gamma \vdash_{ss} M : \text{exp}}{\Gamma \vdash_{ss} \text{pred } M : \text{exp}} \\
(\text{cond}) \frac{\Gamma \vdash_{ss} M : \text{exp} \quad \Gamma \vdash_{ss} N_1 : \text{exp} \quad \Gamma \vdash_{ss} N_2 : \text{exp}}{\Gamma \vdash_{ss} \text{cond } M \, N_1 \, N_2} \quad (\text{rec}) \frac{\Gamma \vdash_{ss} M : A \rightarrow A}{\Gamma \vdash_{ss} Y_A M : A}
\end{array}$$

Imperative constructs

$$\begin{array}{l}
(\text{seq}) \frac{\Gamma \vdash_{ss} M : \text{com} \quad \Gamma \vdash_{ss} N : A}{\Gamma \vdash_{ss} \text{seq}_A M \, N : A} \quad A \in \{\text{com}, \text{exp}\} \\
(\text{assign}) \frac{\Gamma \vdash_{ss} M : \text{var} \quad \Gamma \vdash_{ss} N : \text{exp}}{\Gamma \vdash_{ss} \text{assign } M \, N : \text{com}} \quad (\text{deref}) \frac{\Gamma \vdash_{ss} M : \text{var}}{\Gamma \vdash_{ss} \text{deref } M : \text{exp}} \\
(\text{new}) \frac{\Gamma, x : \text{var} \vdash_{ss} M : A}{\Gamma \vdash_{ss} \text{new } x \text{ in } M : A} \quad A \in \{\text{com}, \text{exp}\} \\
(\text{mkvar}) \frac{\Gamma \vdash_{ss} M_1 : \text{exp} \rightarrow \text{com} \quad \Gamma \vdash_{ss} M_2 : \text{exp}}{\Gamma \vdash_{ss} \text{mkvar } M_1 \, M_2 : \text{var}}
\end{array}$$

Table 3.6: Formation rules for strongly safe IA

3.5.2.2 Safe IA

It turns out that the definition of strongly safe IA is too restrictive: it is possible to identify a larger fragment in which so-called “No-variable-capture” lemma holds. Consider the following IA term:

$$\vdash \text{new } x \text{ in } \lambda z^{\text{exp}}. \text{deref } \underline{x} : \text{exp} \rightarrow \text{exp} .$$

It is not strongly safe, because the variables $x : \text{var}$ and $z : \text{exp}$ have the same order but they are not abstracted together. But since x is a block-allocated variable, no term can ever be substituted for such variables when performing reduction. Therefore there is no point in constraining occurrences of such a variable in the term. So morally, the previous term should be considered safe.

We will therefore distinguish two kinds of variables in a closed term: the “standard ones”—those that are bound by λ -abstractions—and the “imperative” ones—those that are declared by a block-allocation construct—and we will change the side-condition of the abstraction rule so that only variables of the first kind are constrained.

It is also possible to relax the safety constraint for another class of variables. Among the lambda-bound variables, we consider the subclass of variables that are bound by a lambda node λx^{exp} inside a term of the form $\text{mkvar}(\lambda x^{\text{exp}}.M)N$. We call these variables *mkvar-bound variables*. It turns out that it is also possible to relax the safety constraint for this class of variables. To see why this is the case, we need to redefine the typing rules for the **mkvar** construct: We replace the typing in two steps (first abstracting x in M and then constructing $\text{mkvar}(\lambda x^{\text{exp}}.M)N$) by a single step typing rule forming $\text{mkvar}(\lambda x^{\text{exp}}.M)N$ directly from M and N . These two ways of typing the **mkvar** construct are semantically equivalent because it is always possible to eta-expand the first argument of **mkvar** into a term of the form $\lambda x^{\text{exp}}.M$.

The small step semantics is then redefined by replacing the rule

$$\text{assign}(\text{mkvar}MN) n \rightarrow Mn$$

by

$$\text{assign}(\text{mkvar}(\lambda x^{\text{exp}}.M)N) n \rightarrow M[n/x] \quad (3.4)$$

This change ensures that no substitution will ever be done on the term $\lambda x.M$. Hence there is no need to require the term $\lambda x.M$ to be safe: it is sufficient to have that M is safe.

These remarks lead us a more general notion of safety for IA. We consider new judgments of the form $\Gamma | \Xi \vdash_s M : A$, called a *split terms-in-context* (this terminology is borrowed from Abramsky and McCusker’s tutorial on game semantics [AM98b]), where the context is partitioned into two *disjoint* components: The first component Γ will contain the lambda-bound variables that are constrained by the safety restriction; the second component will contain block-declared variables as well as **mkvar**-bound variables. The component Ξ contains variables of type **var** and **exp** only, while the other component can contain variables of any type including **var**. It is straightforward to redefine the typing rule of IA in such a way that these two distinct contexts are maintained appropriately. In particular:

- (i) the abstraction rules can only abstract variables from the first component of the context;
- (ii) the **new** and **mkvar** constructs can only bind variables from the second context component;
- (iii) the side-condition in the abstraction rules constrains only variables from the first context component.

The typing system for this new judgement is given in Table 3.7; The circled rules highlight the important changes from the rules of Table 3.6. A split-term with an empty context Ξ is called a *semi-closed split-term*. We define *safe IA* to be the set of *semi-closed* split-terms typable with the system of rules of Table 3.7. For convenience we introduce the additional rule

$$\frac{\Gamma | \emptyset \vdash_s M : A}{\Gamma \vdash_s M : A}$$

so that safe IA is given by the set of terms-in-context $\Gamma \vdash_s M : A$.

Functional part

$$\begin{array}{c}
(\text{var}^{\text{var}}) \frac{}{\emptyset | \Xi \vdash_s x : \text{var}} \quad x : \text{var} \in \Xi \quad (\text{var}^{\text{exp}}) \frac{}{\emptyset | \Xi \vdash_s x : \text{exp}} \quad x : \text{exp} \in \Xi \\
\\
(\text{var}) \frac{}{\Gamma | \emptyset \vdash_s x : A} \quad x : A \in \Gamma \quad \left((\text{wk}) \frac{\Gamma | \Xi \vdash_s s : A}{\Gamma' | \Xi \vdash_s s : A} \quad \Gamma \subset \Gamma' \wedge \text{dom}(\Gamma') \cap \text{dom}(\Xi) = \emptyset \right) \\
\\
(\delta) \frac{\Gamma \vdash_s M : A}{\Gamma \Vdash_{\text{app}} M : A} \quad (\text{app}_{\text{as}}) \frac{\Gamma | \Xi \vdash_s s : (A_1, \dots, A_n, B) \quad \Gamma | \Xi \vdash_s t_1 : A_1 \dots \Gamma | \Xi \vdash_s t_n : A_n}{\Gamma | \Xi \Vdash_{\text{app}} st_1 \dots t_n : B} \\
\\
\left((\text{app}) \frac{\Gamma | \Xi \vdash_s s : (A_1, \dots, A_n, B) \quad \Gamma | \Xi \vdash_s t_1 : A_1 \dots \Gamma | \Xi \vdash_s t_n : A_n}{\Gamma | \Xi \vdash_s st_1 \dots t_n : B} \quad \text{ord } B \leq \text{ord } \Gamma \right. \\
\\
\left. (\text{abs}) \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n | \Xi \Vdash_{\text{app}} s : B}{\Gamma | \Xi \vdash_s \lambda x_1 \dots x_n. s : (A_1, \dots, A_n, B)} \quad \text{ord } (A_1, \dots, A_n, B) \leq \text{ord } \Gamma \right)
\end{array}$$

Arithmetic and recursion

$$\begin{array}{c}
(\text{const}) \frac{}{\emptyset | \emptyset \vdash_s n : \text{exp}} \quad (\text{succ}) \frac{\Gamma | \Xi \vdash_s M : \text{exp}}{\Gamma | \Xi \vdash_s \text{succ } M : \text{exp}} \quad (\text{pred}) \frac{\Gamma | \Xi \vdash_s M : \text{exp}}{\Gamma | \Xi \vdash_s \text{pred } M : \text{exp}} \\
\\
(\text{cond}) \frac{\Gamma | \Xi \vdash_s M : \text{exp} \quad \Gamma | \Xi \vdash_s N_1 : \text{exp} \quad \Gamma | \Xi \vdash_s N_2 : \text{exp}}{\Gamma | \Xi \vdash_s \text{cond } M \ N_1 \ N_2} \quad (\text{rec}) \frac{\Gamma | \Xi \vdash_s M : A \rightarrow A}{\Gamma | \Xi \vdash_s Y_A M : A}
\end{array}$$

Imperative constructs

$$\begin{array}{c}
(\text{seq}) \frac{\Gamma | \Xi \vdash_s M : \text{com} \quad \Gamma | \Xi \vdash_s N : A}{\Gamma | \Xi \vdash_s \text{seq}_A M \ N : A} \quad A \in \{\text{com}, \text{exp}\} \\
\\
(\text{assign}) \frac{\Gamma | \Xi \vdash_s M : \text{var} \quad \Gamma | \Xi \vdash_s N : \text{exp}}{\Gamma | \Xi \vdash_s \text{assign } M \ N : \text{com}} \quad (\text{deref}) \frac{\Gamma | \Xi \vdash_s M : \text{var}}{\Gamma | \Xi \vdash_s \text{deref } M : \text{exp}} \\
\\
\left((\text{new}) \frac{\Gamma | \Xi, x : \text{var} \vdash_s M : A}{\Gamma | \Xi \vdash_s \text{new } x \text{ in } M : A} \quad A \in \{\text{com}, \text{exp}\} \right. \\
\\
\left. (\text{mkvar}) \frac{\Gamma | \Xi, x : \text{exp} \vdash_s M_1 : \text{exp} \rightarrow \text{com} \quad \Gamma | \Xi \vdash_s M_2 : \text{exp}}{\Gamma | \Xi \vdash_s \text{mkvar } (\lambda x^{\text{exp}}. M_1) \ M_2 : \text{var}} \right)
\end{array}$$

Table 3.7: Formation rules for safe IA.

Example 3.5.2. Strongly safe IA is a subset of safe IA. The following example shows that the inclusion is strict:

$$\begin{aligned} & \vdash_s \lambda f^{(\text{exp} \rightarrow \text{com}) \rightarrow \text{exp}}. \text{new } i \text{ in } f(\lambda x^{\text{exp}}. \text{assign } i \ x) : \text{exp} \\ & \text{but } \not\vdash_{ss} \lambda f^{(\text{exp} \rightarrow \text{com}) \rightarrow \text{exp}}. \text{new } i \text{ in } f(\lambda x^{\text{exp}}. \text{assign } i \ x) : \text{exp} . \end{aligned}$$

It is not strongly safe because the variables i and x are of the same order but only x is abstracted by the lambda. It is safe because unsafe occurrences of block-allocated variables such as i are tolerated in safe IA.

Example 3.5.3. The following term is a safe IA beta-normal term:

$$f : ((\text{exp} \rightarrow \text{exp}) \rightarrow \text{com}) \vdash_s \text{mkvar } (\lambda x^{\text{exp}}. f(\lambda y^{\text{exp}}. \underline{x})) \ 0 : \text{com}^\omega \times \text{exp} .$$

Observe that the unsafe occurrence of the variable x is tolerated because it is a **mkvar**-bound variable.

Since in split safe IA terms, only the variables from the left context component are constrained by the safety restriction, thus the basic property of the safe lambda calculus (Lemma 3.1.2) becomes:

Lemma 3.5.1. *Suppose $\Gamma \mid \Xi \vdash_s M : A$. Then*

$$\forall z : A \in \Gamma. z \in FV(M) \implies \text{ord } z \geq \text{ord } A .$$

The small-step reduction semantics of safe IA is defined similarly as in Sec. 2.1.10 except that β -reduction is replaced by safe β -reduction and the rules for **mkvar** are redefined according to (3.4). Again it is easy to see that safety is preserved by the small-step reduction of IA:

Lemma 3.5.2 (Reduction preserves safety). *Let M be an IA term and \rightarrow denotes the small-step reduction of safe IA. Then $\Gamma \mid \Xi \vdash_s M : A \wedge M \rightarrow N \implies \Gamma \mid \Xi \vdash_s N : A$.*

The proof is by an easy induction.

3.5.2.3 No-variable capture lemma

In which sense are the two calculi above-defined “safe”? In the lambda calculus fragment, the term “safe” refers to the fact that under the *safe typing convention*, substitution can be performed *capture-permitting*. Unfortunately, as we have observed before, in the presence of block-allocation constructs this lemma does not hold anymore because the block-allocation construct **new** does not increase the order of the term that is being formed contrary to λ -abstractions—a property that is crucially used in the proof of the No-variable-capture lemma. The following examples illustrate this. Consider the terms:

$$\begin{aligned} M_1 &\equiv \text{new } x \text{ in seq } (\text{assign } x \ 1) ((\lambda y^0. \text{new } x \text{ in } y)(\text{deref } x)) \\ M_2 &\equiv \lambda x^1. (\lambda y^1. (\text{new } x \text{ in } y \ 0))x \\ M_3 &\equiv \lambda f^2. \text{new } x \text{ in } (\lambda y^1. f(\lambda x^0. y))(\lambda z^0. \text{deref } \underline{x}) \\ M_4 &\equiv \lambda x^{\text{com}}. (\lambda y^{\text{com}}. \text{mkvar } (\lambda x^{\text{exp}}. y) \ 0)x \end{aligned}$$

where the type n is an abbreviation for n_{exp} for $n \in \mathbb{N}$.

All these terms are safe IA terms (but only M_1 and M_2 are strongly safe) and contracting the redexes in those terms using capture-permitting substitution causes problematic variable captures:

- (i) For M_1 , performing the substitution without renaming variables afresh causes the variable x to get captured by the innermost **new** x giving **new** x in seq (assign x 1) (**new** x in deref x) which is observationally equivalent to 0 (since block-allocated variables are initialized with 0). On the other hand standard substitution gives: **new** x in seq (assign x 1) (**new** z in deref x) which is observationally equivalent to 1.

- (ii) For M_2 , the capture-permitting substitution gives $\lambda x^1.(\text{new } x \text{ in } x \ 0)$ which is not even a typable in IA;
- (iii) For M_3 , capture-permitting substitution gives $\lambda f^2.\text{new } x \text{ in } \lambda y^1.f(\lambda x^0.(\lambda x^0.\text{deref } x))$ which is not a typable IA term;
- (iv) Finally for M_4 , capture-permitting substitution gives $(\lambda y^{\text{com}}.\text{mkvar } (\lambda x^{\text{exp}}.x) \ 0)$ which, again, is not a typable IA term because the subterm $\lambda x^{\text{exp}}.x$ is of type $\text{exp} \rightarrow \text{exp}$ instead of $\text{exp} \rightarrow \text{com}$.

To deal with the first two examples, we have no other choice than renaming block-declared variables afresh upon substitution. For the last two kinds of variable capture (which only happen for safe terms that are not strongly safe) we can resolve the problem by adopting the following convention:

CONVENTION 3.5.1 The set of names used for block-declared and **mkvar**-bound variables is disjoint from the set of names used for lambda-abstracted variables. This convention can be enforced by tagging each variable occurrence to indicate whether it is a block-allocated variable or a lambda-abstracted variable, thus permitting one to resolve any binding ambiguity. Observe that this convention is stronger than requiring that the sets of names of the two context components of a split-term are disjoint because this only constrains the free variables of the term whereas what we are requiring here is a global constraint on all variables names occurring in the term including the bound ones.

This leads us to the following notion of substitution which performs variable renaming only for block-allocated variables and **mkvar**-bound variables:

Definition 3.5.1. The *semi-capture-permitting substitution* of the term-in-context $\Gamma|\Xi \vdash N : A$ for x in the term-in-context $\Gamma, x : A|\Xi \vdash M : B$, written $\Gamma|\Xi \vdash M\{N/x\}$, is defined inductively on M as follows:

$$\begin{aligned}
x\{N/x\} &= N \\
y\{N/x\} &= y & y \neq x; \\
(\lambda x^\tau.M)\{N/x\} &= \lambda x^\tau.M \\
(\lambda y^\tau.M)\{N/x\} &= \lambda y^\tau.M\{N/x\} & \text{where } y \neq x; \\
(\text{new } x \text{ in } M)\{N/x\} &= \text{new } x \text{ in } M \\
(\text{new } y \text{ in } M)\{N/x\} &= \text{new } z \text{ in } M\{z/y\}\{N/x\} & \text{if } x \neq y, z \text{ fresh;} \\
(\text{mkvar } (\lambda x^\tau.M_1) M_2)\{N/x\} &= \text{mkvar } (\lambda x^\tau.M_1) M_2\{N/x\} \\
(\text{mkvar } (\lambda y^\tau.M_1) M_2)\{N/x\} &= \text{mkvar } (\lambda z^\tau.M_1\{z/y\}\{N/x\}) M_2\{N/x\} & \text{if } x \neq y, z \text{ fresh.}
\end{aligned}$$

The other constants and application cases are defined inductively in the standard way.

It is now possible to state a version of the *No-variable capture lemma* for safe IA:

Lemma 3.5.3 (No-variable capture). *Suppose that $\Gamma|\Xi \vdash_s N : A$ and $\Gamma, x : A|\Xi \vdash_s M : B$ then*

- (i) *under convention 3.1.2 and 3.5.1, the substitution $M[N/x]$ can be performed semi-capture-permitting:*

$$M[N/x] \equiv M\{N/x\}.$$

- (ii) *under convention 3.5.1 and if $\Gamma|\Xi \vdash M\{N/x\} : B$ is a valid (not-necessarily safe) IA judgement then $M[N/x] \equiv M\{N/x\}$.*

The proof is a trivial extension of Lemma 3.1.4 and 3.1.5.

Corollary 3.5.2. *Let $\Gamma \vdash_s N : A$ and $\Gamma, x : A \vdash_s M : B$ be safe IA terms-in-context.*

- (i) *If convention 3.1.2 is adopted then $M[N/x] \equiv M\{N/x\}$;*
- (ii) *If $\Gamma \vdash M\{N/x\} : B$ is typable in IA then $M[N/x] \equiv M\{N/x\}$.*

3.5.3 Generalization to other applied lambda calculi

In this section, we define the notion of safety for any given applied lambda calculus extended with a stock of interpreted constants Σ but without recursion. The syntax of the language is given by some system of rules producing split-term of the form

$$\Gamma \mid \Xi \vdash M : T$$

for some simple-type T , where variables in the context Γ and Ξ are called the Γ -variables and Ξ -variables respectively. The calculus must verify the following prerequisites:

- (i) The abstraction rule can only abstract Γ -variables;
- (ii) The terms of the languages are given by the semi-closed split-terms $\Gamma \mid \emptyset \vdash M : T$ abbreviated as $\Gamma \vdash M : T$.

Consequently, a Ξ -variable can only be “bound” by some constant construct of the language but not by a lambda-abstraction.

Definition 3.5.2. Consider an applied lambda calculus as defined above. Its *safe fragment* is defined as the system obtained by restricting the pure lambda calculus fragment of the language in such a way that:

- (i) the restriction of the system to its pure simply-typed fragment coincides with the definition of the safe lambda calculus;
- (ii) The side-condition of the abstraction and application rules constrains only Γ -variables.

Terms-in-context thus generated are written $\Gamma \vdash_s M : T$.

An immediate consequence is that terms-in-context of the safe fragment verify the basic property of the safe lambda calculus:

$$\Gamma \vdash_s M : T \implies \forall z : A \in \Gamma. z \in FV(M) \implies \text{ord } A \geq \text{ord } T .$$

Further, in order for this language to be of any use, it must verify the subject reduction lemma (*i.e.*, the small-step reduction semantics must preserve safety).

The results of the previous sections show that IA and the recursion-free fragments of PCF both fit in this setting.

3.6 Related work

The safety condition for higher-order grammars

We have mentioned the result of Knapik et al. [KNU02] that infinite trees generated by *safe* higher-order grammars have decidable MSO theories. A natural question to ask is whether the *safety condition* is really necessary. This has been partially answered by Aehlig et al. [AdMO05a] where it was shown that safety is not a requirement at level 2 to guarantee MSO decidability. Also, for the restricted case of word languages, the same authors have shown [AdMO05b] that level 2 safe higher-order grammars are as powerful as (non-deterministic) unsafe ones. De Miranda’s thesis [dM06] proposes a unified framework for the study of higher-order grammars and gives a detailed analysis of the safety constraint at level 2.

More recently, Ong obtained a more general result and showed that the MSO theory of infinite trees generated by higher-order grammars of any level, *whether safe or not*, is decidable [Ong06a]. Using an argument based on innocent game-semantics, he establishes a correspondence between the *computation tree* of a higher-order grammar and the *value tree* that it generates: Paths in the *value tree* correspond to P-views of traversals of the computation tree. Decidability is then obtain by reducing the problem to the acceptance of the (annotated) computation tree by a certain alternating parity tree automaton.

The equivalence of *safe* higher-order grammars and higher-order deterministic push-down automata for the purpose of generating infinite trees [KNU02] has its counterpart in the general (not necessarily safe) case: the paper [HMOS08] establishes the equivalence of order- n higher-order

grammars and order- n *collapsible pushdown automata*. Those automata form a new kind of push-down systems in which every stack symbol has a link to a stack situated somewhere below it and with an additional stack operation whose effect is to “collapse” a stack s to the state indicated by the link from the top stack symbol.

Chapter 4

A Concrete Presentation of Game Semantics

As observed before, analyzing the effect that a syntactic restriction (such as safety) has on the game semantic model is a difficult task since the main feature of game semantics is precisely to be syntax-independent. The aim of this chapter is to establish an explicit correspondence between the game denotation of a term and its syntax. This will be used in the next chapter to give a characterization of the game semantics of the safe lambda-calculus.

Our approach follows ideas recently introduced by Ong [Ong06a], namely the notion of *computation tree* of a simply-typed lambda-term and *traversals* over the computation tree. A computation tree is just an abstract syntax tree (AST) representation of the η -long normal form of a term. Traversals are justified sequences of nodes of the computation tree respecting some formation rules. They are meant to describe the computation of the term, but at the same time they carry information about the syntax of the term in the following sense: the *P-view* of a traversal (computed in the same way as P-view of plays in game semantics) is a path in the computation tree. Traversals provide a way to perform *local computation* of β -reductions as opposed to a global approach where β -redexes are contracted using substitution.

The culmination of this chapter is the *Correspondence Theorem* (Theorem 4.2.2). It states that traversals over the computation tree are just representations of the uncovering of plays in the strategy-denotation of the term. Hence there is an isomorphism between the strategy denotation of a term and its revealed game denotation. In a nutshell, the revealed denotation is computed similarly to the standard strategy denotation except that internal moves are not hidden after composition. In order to make a connection with the standard game denotation, we define a *reduction* operation on traversals that eliminates occurrences of “internal nodes”. These node occurrences are the counterparts of internal moves that are hidden when performing strategy composition in game semantics. This leads to a correspondence between the standard game denotation of a term and the set of reductions of traversals over its computation tree.

The traversal reduction transformation is such that the syntactical content of traversals is not entirely lost after reduction. Together with the Correspondence theorem this makes it possible to analyze the effect that a syntactic restriction has on the strategy denoting a term. This is illustrated in the next chapter where we make use of the Correspondence Theorem to analyze the game semantics of the safety restriction.

Related works: The useful transference technique between plays and traversals was originally introduced by Ong for studying the decidability of monadic second-order theories of infinite structures generated by higher-order grammars [Ong06a]. In this setting, the Σ -constants or terminal symbols are at most order 1, and are *uninterpreted*. Here we present an extension of this framework to the general case of the simply typed lambda calculus with free variables of any order. Further the term considered is not required to be of ground type contrary to higher-order grammars. This requires us to add new traversal rules to handle variable whose value is undetermined (*i.e.*, those

that cannot be resolved by contracting a redex). We also accommodate computation trees with additional nodes accounting for answer moves of game semantics. This enables our framework to be extended to languages with interpreted constants such as PCF and Idealized Algol.

A notion of local computation of β -reduction has also been investigated through the use of special graphs called “virtual nets” that embed the lambda calculus [DR93].

Asperti et al. introduced [ADLR94] a notion of graph based on Lamping’s graphs [Lam90] to represent lambda-terms. The authors unify different notions of paths (regular, legal, consistent and persistent paths) that have appeared in the literature as ways to implement graph-based reduction of lambda-expressions. We can regard a traversal as an alternative notion of path adapted to the graph representation of λ -expressions given by computation trees.

4.1 Computation tree

We work in the general setting of the simply typed lambda-calculus extended with a fixed set Σ of higher-order uninterpreted constants.¹ We fix a simply typed term-in-context $\Gamma \vdash M : T$ for the rest of the section.

4.1.1 Definition

We define the *computation tree* of a simply-typed lambda-term as an abstract syntax tree representation of its η -long normal form (Def. 3.1.7). Our definition generalizes the notion of computation tree for higher-order grammar [Ong06a].

We recall that a term M in η -normal form is of the form $\lambda \bar{x}. s_0 s_1 \dots s_m$ where $\bar{x} = x_1 \dots x_n$ for $n \geq 0$ and $s_0 s_1 \dots s_m$ is of ground type, each s_j for $j \in 1..m$ is in η -long nf, and either s_0 is a variable or a constant and $m \geq 0$; or s_0 is an abstraction $\lambda \bar{y}. s$ and $m \geq 1$ where s is in η -long nf. If M is of ground type (i.e., $T = o$) then its η -long nf is of the form $\lambda.N$; The symbol ‘ λ ’ does not correspond to a real lambda-abstraction—we call it ‘dummy lambda’—but it will be convenient to keep it in expressions representing eta-long normal forms.

Definition 4.1.1. Let $\Gamma \vdash_{\text{st}} M : T$ be a simply typed term with variable names ranging in \mathcal{V} and constants in Σ . The *pre-computation tree* $\tau^-(M)$ with labels taken from $\{\emptyset\} \cup \Sigma \cup \mathcal{V} \cup \{\lambda x_1 \dots x_n \mid x_1, \dots, x_n \in \mathcal{V}, n \in \mathbb{N}\}$, is defined inductively on its η -long form as follows.

$$\begin{aligned} \text{For } m \geq 0, z \in \mathcal{V} \cup \Sigma: \tau^-(\lambda \bar{x}. z s_1 \dots s_m : o) &= \lambda \bar{x} \langle z \langle \tau(s_1), \dots, \tau(s_m) \rangle \rangle \\ \text{for } m \geq 1: \tau^-(\lambda \bar{x}. (\lambda \bar{y}. t) s_1 \dots s_m : o) &= \lambda \bar{x} \langle @ \langle \tau(\lambda \bar{y}. t), \tau(s_1), \dots, \tau(s_m) \rangle \rangle, \end{aligned}$$

where we write $l \langle t_1, \dots, t_n \rangle$ for $n \geq 0$ to denote the *ordered tree* whose root is labelled l and has n child-subtrees t_1, \dots, t_n . These trees are illustrated in Table 4.1.

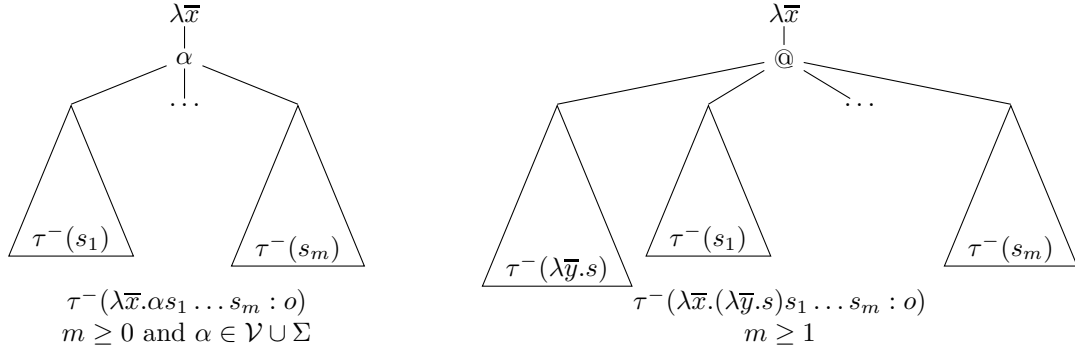
By convention the first level of a tree (where the root lies) is numbered 0. In the tree $\tau^-(M)$, nodes at odd-levels are variable or application nodes, and at even-levels lies the λ -nodes. A single λ -node can represent several consecutive abstractions or it can just be a *dummy lambda* (in which case the corresponding subterm is of ground type).

Definition 4.1.2. Let M be a simply typed term not necessarily in η -normal form. Let \mathcal{D} denote the set of values of base type o . The *computation tree* of M , written $\tau(M)$ is the tree obtained from $\tau^-(\lceil M \rceil)$ by attaching leaves to each node as follows: for every node $n \in \tau^-(M)$, the corresponding node in $\tau(\lceil M \rceil)$ has a child leaf labelled v_n , called *value-leaf*, for every possible value $v \in \mathcal{D}$.

The inner nodes of the tree are of three kinds:

- λ -nodes labelled $\lambda \bar{x}$ (note that a λ -node represents several consecutive variable abstractions),

¹A constant f is *uninterpreted* if the small-step semantics of the language does not contain any rule of the form $f \dots \rightarrow e$. f can be regarded as a data constructor.



- application nodes labelled @,
- variable or constant nodes labelled α for some constant or variable α .

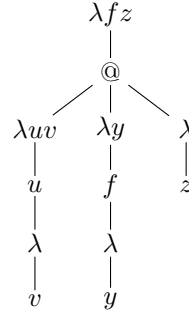
A node is said to be **prime** if it is the 0^{th} child of an @-node. An inner node whose parent is a @-nodes or a Σ -node is called a **spawn** node.

Example 4.1.1. Take $\vdash_{\text{st}} \lambda f^{o \rightarrow o}.(\lambda u^{o \rightarrow o}.u)f : (o \rightarrow o) \rightarrow o \rightarrow o$.

Its η -long normal form is:

$$\begin{aligned} &\vdash_{\text{st}} \lambda f^{o \rightarrow o} z^o. \\ &\quad (\lambda u^{o \rightarrow o} v^o. u(\lambda v)) \\ &\quad (\lambda y^o. f y) \\ &\quad (\lambda z) \\ &: (o \rightarrow o) \rightarrow o \rightarrow o \end{aligned}$$

Its computation tree is:

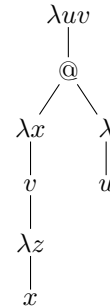


Example 4.1.2. Take $\vdash_{\text{st}} \lambda u^o v^{((o \rightarrow o) \rightarrow o)}.(\lambda x^o. v(\lambda z^o. x))u : o \rightarrow ((o \rightarrow o) \rightarrow o) \rightarrow o$.

Its η -long normal form is:

$$\begin{aligned} &\vdash_{\text{st}} \lambda u^o v^{((o \rightarrow o) \rightarrow o)}. \\ &\quad (\lambda x^o. v(\lambda z^o. x))u \\ &: o \rightarrow ((o \rightarrow o) \rightarrow o) \rightarrow o \end{aligned}$$

Its computation tree is:



NOTATIONS 4.1.1 We write \otimes to denote the root of $\tau(M)$. We write E to denote the parent-child relation of the tree, V for the set of vertices (leaves and inner nodes) of the tree, N for the set of inner nodes and L for the set of value-leaves. Thus $V = N \cup L$.

We write N_Σ for the set of Σ -labelled nodes, N_\circ for the set of \circ -labelled nodes, N_{var} for the set of variable nodes, N_{fv} for the subset of N_{var} constituted of free-variable nodes, N_{prime} for the set of prime nodes and N_{spawn} for the set of spawn nodes ($= N \cap E(N_\circ \cup N_\Sigma)$).

For $\$$ ranging in $\{\circ, \lambda, \text{var}, \text{fv}\}$, we write $L_\$$ to denote the set of value-leaves of nodes from $N_\$$ and $V_\$$ to denote the set of nodes of $N_\$$ or value-leaves of nodes of $N_\$$; formally $L_\$ = \{v_n \mid n \in N_\$, v \in \mathcal{D}\}$ and $V_\$ = N_\$ \cup L_\$$.

For any lambda node n in N_λ we write $M^{(n)}$ for the subterm rooted at n and $V^{(n)}$ for the set of nodes and leaves of the sub-computation tree $\tau(M^{(n)})$; formally $V^{(n)} = E^*(\{n\})$.

Let \mathcal{T} denote the set of lambda-terms. Each subtree of the computation tree $\tau(M)$ represents a subterm of $\llbracket M \rrbracket$. We define the function $\kappa : N \rightarrow \mathcal{T}$ that maps a node $n \in N$ to the subterm of $\llbracket M \rrbracket$ corresponding to the subtree of $\tau(M)$ rooted at n . In particular $\kappa(r) = \llbracket M \rrbracket$.

Definition 4.1.3 (Type and order of a node). Suppose $\Gamma \vdash M : T$. The **type** of an inner-node $n \in N$ of $\tau(M)$ written $\text{type}(n)$ is defined as follows:

$$\begin{aligned} \text{type}(\circ) &= \Gamma \rightarrow T, \\ \text{for } n \in (N_\lambda \cup N_\circ) \setminus \{\circ\}: \text{type}(n) &= \text{type of the term } \kappa(n), \\ \text{for } n \in N_{\text{var}} \cup N_\Sigma: \text{type}(n) &= \text{type of the variable labelling } n. \end{aligned}$$

where the notation $\Gamma \rightarrow T$ is an abbreviation for (A_1, \dots, A_p, T) where A_1, \dots, A_p are the types of the variables in the context Γ .

The **order** of a node n , written $\text{ord } n$, is defined as follows: a value-leaf $v \in L$ has order 0 and the order of an inner node $n \in N$ is defined as the order of its type.

In particular, the type of a lambda node different from the root is the type of the term represented by the sub-tree rooted at that node, and the type of a variable-node is the type of the variable labelling it.

Since the computation tree is calculated from the η -long normal form, all the \circ -nodes have order 0 ($\text{ord } \circ = 0$); for any lambda node $\lambda \bar{\xi} \neq \circ$ we have $\text{ord } \lambda \bar{\xi} = 1 + \max_{z \in \bar{\xi}} \text{ord } z$; and if the root \circ is labelled $\lambda \bar{\xi}$ then $\text{ord } \circ = 1 + \max_{z \in \bar{\xi} \cup \Gamma} \text{ord } z$ with the convention $\max \emptyset = -1$.

REMARK 4.1.1

- In a computation tree, nodes at even level are λ -nodes and nodes at odd level are either application nodes, variable or constant nodes;

- for any ground type variable or constant α , λ ;

$$\begin{array}{c} \lambda \\ | \\ \alpha \end{array}$$

- for any higher-order variable or constant $\alpha : (A_1, \dots, A_p, o)$, the computation tree $\tau(\alpha)$ has the following form:

$$\begin{array}{c} \lambda \\ \alpha \\ \swarrow \quad \vdots \quad \searrow \\ \lambda \bar{\xi}_1 \quad \dots \quad \lambda \bar{\xi}_p \\ \dots \quad \dots \quad \dots \end{array}$$

- for any tree of the form $\lambda \bar{\varphi}$, we have $\text{ord } \kappa(n) = 0$.

$$\begin{array}{c} \lambda \bar{\varphi} \\ n \\ \swarrow \quad \vdots \quad \searrow \\ \lambda \bar{\xi}_1 \quad \dots \quad \lambda \bar{\xi}_p \\ \dots \quad \dots \quad \dots \end{array}$$

Definition 4.1.4 (Binder). We say that a variable node n labelled x is **bound** by a node m , and m is called the **binder** of n , if m is the closest node in the path from n to the root such that m is labelled $\lambda \bar{\xi}$ with $x \in \bar{\xi}$.

4.1.2 Pointers and justified sequence of nodes

4.1.2.1 Definitions

Definition 4.1.5 (Enabling). The *enabling relation* \vdash is defined on the set of nodes and leaves of the computation tree as follows. We write $m \vdash n$ and we say that m enables n if and only if $m \in L \cup N_\lambda \cup N_{\text{var}}$ and one of the following conditions holds:

- $n \in N_{\text{fv}}$ and m is the root \otimes ;
- $n \in N_{\text{var}} \setminus N_{\text{fv}}$ and m is n 's binder, in which case we write $m \vdash_i n$ to precise that n is the i^{th} variable bound by m ;
- $n \in N_\lambda$ and m is n 's parent;
- $n \in L$ and m is n 's parent (i.e., $n = v_m$ for some $v \in \mathcal{D}$).

Formally:

$$\begin{aligned} \vdash = & \{(\otimes, n) \mid n \in N_{\text{fv}}\} \\ & \cup \{(\lambda \bar{x}, x) \mid x \in N_{\text{var}} \setminus N_{\text{fv}} \wedge \lambda \bar{x} \text{ is } x\text{'s binder}\} \\ & \cup \{(m, \lambda \bar{\eta}) \mid m \text{ is } \lambda \bar{\eta}\text{'s parent and } \lambda \bar{\eta} \in N_\lambda\} \\ & \cup \{(m, v_m) \mid v \in \mathcal{D}, m \in N\} \end{aligned}$$

Note that in particular, free variable nodes are enabled by the root. Table 4.2 recapitulates the possible node types for the enabler node depending on the type of n .

If $n \in _$	then $m \in _$
N_λ	$N_{\text{var}} \cup N_\Sigma \cup N_\otimes$
L_{var}	N_{var}
L_\otimes	N_\otimes
L_Σ	N_Σ
N_{var}	N_λ
N_Σ	n.a.
N_\otimes	n.a.
L_λ	N_λ

Table 4.2: Type of the enabler node in “ $m \vdash n$ ”.

We say that a node n_0 of the computation tree is *hereditarily enabled* by $n_p \in N$ if there are nodes $n_1, \dots, n_{p-1} \in N$ such that n_{i+1} enables n_i for all $i \in 0..p-1$.

For any set of nodes $S, H \subseteq N$ we write $S^{H\vdash}$ to denote the subset $S \cap \vdash^*(H)$ of S constituted of nodes hereditarily enabled by some node in H . Formally:

$$S^{H\vdash} = \{n \in S \mid \exists n_0 \in H \text{ s.t. } n_0 \vdash^* n\}.$$

If H is a singleton $\{n_0\}$ then we abbreviate $S^{\{n_0\}\vdash}$ into $S^{n_0\vdash}$.

We have $V_{\text{var}}^{\otimes\vdash} = V \setminus (V_{\text{var}}^{N_\otimes\vdash} \cup V_{\text{var}}^{N_\Sigma\vdash})$. The element of $N_{\text{var}}^{\otimes\vdash}$ (i.e., variable nodes that are hereditarily enabled by the root of $\tau(M)$) are called *input-variables nodes*.

We use the following numbering conventions: the first child of a \otimes -node—a prime node—is numbered 0; the first child of a variable or constant node is numbered 1; and variables in $\bar{\xi}$ are numbered from 1 onward ($\bar{\xi} = \xi_1 \dots \xi_n$). We write $n.i$ to denote the i th child of node n .

Definition 4.1.6 (Justified sequence of nodes). A *justified sequence of nodes* is a sequence of nodes s of the computation tree $\tau(M)$ with pointers. Each occurrence in s of a node n in $L \cup N_\lambda \cup N_{\text{var}}$ has a link pointing to some preceding occurrence of a node m verifying $m \vdash n$; and occurrences of nodes in $N_\otimes \cup N_\Sigma$ do not have pointer.

If an occurrence n points to an occurrence m in s then we say that m *justifies* n . If n is an inner node then we represent this pointer in the sequence as $\overset{i}{\overleftarrow{m}} \dots n$ where the label indicates that either n is labelled with the i th variable abstracted by the λ -node m or that n is the i th child of m . For leaves, pointers are represented as $\overset{v}{\overleftarrow{m}} \dots v_m$.

Thus a pointer in a justified sequence of nodes has one of the following forms:

$$\begin{aligned}
 & \overset{r}{\overleftarrow{r}} \dots z \quad \text{for some occurrences } r \text{ of } \tau(M)\text{'s root and } z \in N_{fv} ; \\
 \text{or } & \overset{i}{\overleftarrow{\lambda \xi}} \dots \xi_i \quad \text{for some variable } \xi_i \text{ bound by } \lambda \xi, i \in 1..|\overline{\xi}| ; \\
 \text{or } & \overset{j}{\overleftarrow{@}} \dots \lambda \overline{\eta} \quad j \in \{1..(arity(@) - 1)\} ; \\
 \text{or } & \overset{k}{\overleftarrow{\alpha}} \dots \lambda \overline{\eta}, \quad \text{for } \alpha \in N_\Sigma \cup N_{var}, k \in \{1..arity(\alpha)\} ; \\
 \text{or } & \overset{v}{\overleftarrow{m}} \dots v_m \quad \text{for some value } v \in \mathcal{D} .
 \end{aligned}$$

We say that an inner node n in of a justified sequence of nodes is *answered*² by the value-leaf v_n if there is an occurrence of v_n for some value v in the sequence that points to n , otherwise we say that n is *unanswered*. The last unanswered node is called the *pending node*. A justified sequence of nodes is *well-bracketed* if each value-leaf occurring in it is justified by the pending node at that point.

For any justified sequence of nodes and leaves t we write $?(t)$ to denote the subsequence of t consisting only of unanswered nodes. Formally:

$$\begin{aligned}
 ?(u_1 \cdot \overset{v}{\overleftarrow{u_2}} \cdot v_n) &= ?(u_1 \cdot n \cdot u_2) \setminus \{n\} & \text{for some value } v \in \mathcal{D} \\
 ?(u \cdot n) &= ?(u) \cdot n & \text{for } n \notin L
 \end{aligned}$$

where $u \setminus \{n\}$ denotes the subsequence of u obtained by removing the occurrence n .

If u is a well-bracketed sequences then $?(u)$ can be defined as follows:

$$\begin{aligned}
 ?(u \cdot \overset{v}{\overleftarrow{n}} \dots v_n) &= ?(u) & \text{for some value } v \in \mathcal{D} \\
 ?(u \cdot n) &= ?(u) \cdot n & \text{where } n \notin L .
 \end{aligned}$$

NOTATIONS 4.1.2 We write $s = t$ to denote that the justified sequences t and s have same nodes and pointers. Justified sequence of nodes can be ordered using the prefix ordering: $t \leq t'$ if and only if $t = t'$ or the sequence of nodes t is a finite prefix of t' (and the pointers of t are the same as the pointers of the corresponding prefix of t'). Note that with this definition, infinite justified sequences can also be compared. This ordering gives rise to a complete partial order. We say that a node n_0 of a justified sequence is *hereditarily justified* by n_p if there are nodes n_1, n_2, \dots, n_{p-1} in the sequence such that for all $i \in 0..p-1$, n_i points to n_{i+1} . We write t^ω to denote the last occurrence of t and $ip(t)$ for the immediate prefix of t obtained by removing t 's last occurrence. We write $jp(t)$ for the sequence $t_{\leq j}$ where j is the justifier of t^ω in t .

4.1.2.2 Projection

We define two different projection operations on justified sequences of nodes.

Definition 4.1.7 (Projection on a set of nodes). Let A be a subset of V , the set of nodes and leaves of $\tau(M)$, and t be a justified sequence of nodes then we write $t \upharpoonright A$ for the subsequence of t consisting of nodes in A . This operation can cause a node n to lose its pointer. In that case we reassign the target of the pointer to the last node in $t_{\leq n} \upharpoonright A$ that hereditarily justifies n (this node can be found by following the pointers from n until we reach a node that appear in A). If there is no such node then n just loses its pointer.

²This terminology is deliberately suggestive of the correspondence with game-semantics.

Definition 4.1.8 (Hereditary projection). Let t be a justified sequence of nodes of $\mathcal{Trav}(M)$ and n be some occurrence in t . We define the justified sequence $t \upharpoonright n$ as the subsequence of t consisting of nodes hereditarily justified by n in t .

Lemma 4.1.1. *The projection function $_ \upharpoonright n$ defined on the cpo of justified sequences ordered by the prefix ordering is continuous.*

Proof. Clearly $_ \upharpoonright n$ is monotonous. Suppose that $(t_i)_{i \in \omega}$ is a chain of justified sequences. Let u be a finite prefix of $(\bigvee t_i) \upharpoonright n$. Then $u = s \upharpoonright n$ for some finite prefix s of $\bigvee t_i$. Since s is finite we must have $s \leq t_j$ for some $j \in \omega$. Therefore $u \leq t_j \upharpoonright n \leq \bigvee (t_j \upharpoonright n)$. This is valid for any finite prefix u of $(\bigvee t_i) \upharpoonright n$ thus $(\bigvee t_i) \upharpoonright n \leq \bigvee (t_j \upharpoonright n)$. \square

The nodes occurrences that do not have pointers in a justified sequence are called **initial occurrences**. An initial occurrence is either the root of the computation tree, an @-node or a Σ -node. Let n be occurrence in a justified sequence of nodes t . The subsequence of t consisting of occurrences that are her. just. by the same *initial occurrence* as n is called **thread** of n . Thus each thread in a traversal contains a single initial occurrence. The thread of n is given by $n \upharpoonright i$ where i is the first node in t hereditarily justifying n ; i is called the **initial occurrence of the thread of n** .

4.1.2.3 Views

The notion of **P-view** $\ulcorner t \urcorner$ of a justified sequence of nodes t is defined the same way as the P-view of a justified sequences of moves in Game Semantics:

Definition 4.1.9 (P-view of justified sequence of nodes). The P-view of a justified sequence of nodes t of $\tau(M)$, written $\ulcorner t \urcorner$, is defined as follows:

$$\begin{aligned} \ulcorner \epsilon \urcorner &= \epsilon \\ \ulcorner s \cdot n \urcorner &= \ulcorner s \urcorner \cdot n && \text{for } n \in N_{\text{var}} \cup N_{\Sigma} \cup N_{@} \cup L_{\lambda} ; \\ \ulcorner s \cdot \overbrace{m \dots n} \urcorner &= \ulcorner s \urcorner \cdot \overbrace{m \dots n} && \text{for } n \in L_{\text{var}} \cup L_{\Sigma} \cup L_{@} \cup N_{\lambda} ; \\ \ulcorner s \cdot r \urcorner &= r && \text{if } r \text{ is an occurrence of } \oplus \text{ (}\tau(M)\text{'s root) .} \end{aligned}$$

The equalities in the definition determine pointers implicitly. For instance in the second clause, if in the left-hand side, n points to some node in s that is also present in $\ulcorner s \urcorner$ then in the right-hand side, n points to that occurrence of the node in $\ulcorner s \urcorner$.

The O-view of s , written $\lfloor s \rfloor$, is defined dually.

Definition 4.1.10 (O-view of justified sequence of nodes). The O-view of a justified sequence of nodes t of $\tau(M)$, written $\lfloor t \rfloor$, is defined as follows:

$$\begin{aligned} \lfloor \epsilon \rfloor &= \epsilon \\ \lfloor s \cdot n \rfloor &= \lfloor s \rfloor \cdot n && \text{for } n \in L_{\text{var}} \cup L_{\Sigma} \cup L_{@} \cup N_{\lambda} ; \\ \lfloor s \cdot \overbrace{m \dots n} \rfloor &= \lfloor s \rfloor \cdot \overbrace{m \dots n} && \text{for } n \in N_{\text{var}} \cup L_{\lambda} ; \\ \lfloor s \cdot n \rfloor &= n && \text{for } n \in N_{@} \cup N_{\Sigma} . \end{aligned}$$

We borrow some game semantic terminology:

Definition 4.1.11. A justified sequence of nodes s satisfies:

- **Alternation** if for any two consecutive nodes in s , one is in V_{λ} and not the other one;
- **P-visibility** if for every occurrence in s of a node in $N_{\text{var}} \cup L_{\lambda}$, its justifier occur in the P-view at that point;
- **O-visibility** if the justifier of each lambda node in s occurs in the O-view at that point.

Property 4.1.1. The P-view (resp. O-view) of a justified sequence verifying P-visibility (resp. O-visibility) is a well-formed justified sequence verifying P-visibility (resp. P-visibility).

This is proved by an easy induction.

4.1.3 Traversal of the computation tree

A *traversal* is a justified sequence of nodes of the computation tree where each node indicates a step that is taken during the evaluation of the term.

4.1.3.1 Traversals for simply-typed λ -terms

We first consider the simply typed lambda-calculus without interpreted constants. Everything remains valid in the presence of *uninterpreted* constants as we can just consider them as free variables.

We define the notion of traversal over the computation tree $\tau(M)$. We will then we show how to extend the notion of traversal to more general settings with interpreted constants.

Definition 4.1.12 (Traversals for simply-typed lambda-terms). The set $\mathcal{Trav}(M)$ of *traversals* over $\tau(M)$ is defined by induction over the rules of Table 4.3. A traversal that cannot be extended by any rule is said to be *maximal*.

Initialization rules

(Empty) $\epsilon \in \mathcal{Trav}(M)$.

(Root) The sequence constituted of a single occurrence of $\tau(M)$'s root is a traversal.

Structural rules

(Lam) If $t \cdot \lambda \bar{\xi}$ is a traversal then so is $t \cdot \lambda \bar{\xi} \cdot n$ where n denotes $\lambda \bar{\xi}$'s child and:

- If $n \in N_{@} \cup N_{\Sigma}$ then it has no justifier;
- if $n \in N_{\text{var}} \setminus N_{\text{fv}}$ then it points to the only occurrence^a of its binder in $\ulcorner t \cdot \lambda \bar{\xi} \urcorner$;
- if $n \in N_{\text{fv}}$ then it points to the only occurrence of the root \otimes in $\ulcorner t \cdot \lambda \bar{\xi} \urcorner$.

(App) If $t \cdot @$ is a traversal then so is $t \cdot @ \cdot n$.

Input-variable rules

(InputVar) If t is a traversal where $t^\omega \in N_{\text{var}}^{\otimes+} \cup L_{\lambda}^{\otimes+}$ and x is an occurrence of a variable node in $\ulcorner t \urcorner$ then so is $t \cdot n$ for any child λ -node n of x , n pointing to x .

(InputValue) If $t_1 \cdot x \cdot t_2$ is a traversal with pending node $x \in N_{\text{var}}^{\otimes+}$ then so is $t_1 \cdot x \cdot t_2 \cdot v_x$ for all $v \in \mathcal{D}$.

Copy-cat rules

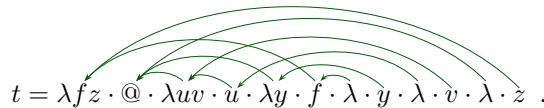
(Var) If $t \cdot n \cdot \lambda \bar{x} \dots x_i$ is a traversal where $x_i \in N_{\text{var}}^{\otimes+}$ then so is $t \cdot n \cdot \lambda \bar{x} \dots x_i \cdot \lambda \bar{\eta}_i$.

(Value) If $t \cdot m \cdot n \dots v_n$ is a traversal where $n \in N$ then so is $t \cdot m \cdot n \dots v_n \cdot v_m$.

Table 4.3: Traversal rules for the simply typed lambda-calculus.

^aProp. 4.1.1 will show that P-views are paths in the tree thus n 's enabler occurs exactly once in the P-view.

Example 4.1.3. The following justified sequence is a traversal of the computation tree of example 4.1.1:



REMARK 4.1.2

1. The rule (Value) from Table 4.3 can be equivalently reformulated into four distinct rules $(\text{Value}^{\lambda \mapsto @})$, $(\text{Value}^{@ \mapsto \lambda})$, $(\text{Value}^{\lambda \mapsto \text{var}})$ and $(\text{Value}^{\text{var} \mapsto \lambda})$, each one dealing with a different possible category for the nodes n and m :

$(\text{Value}^{\lambda \mapsto @})$ If $t \cdot @ \cdot \lambda \bar{z} \dots v_{\lambda \bar{z}}$ is a traversal then so is $t \cdot @ \cdot \lambda \bar{z} \dots v_{\lambda \bar{z}} \cdot v_{@}$.

$(\text{Value}^{@ \mapsto \lambda})$ If $t \cdot \lambda \bar{\xi} \cdot @ \dots v_{@}$ is a traversal then so is $t \cdot \lambda \bar{\xi} \cdot @ \dots v_{@} \cdot v_{\lambda \bar{\xi}}$.

$(\text{Value}^{\lambda \mapsto \text{var}})$ If $t \cdot y \cdot \lambda \bar{\xi} \dots v_{\lambda \bar{\xi}}$ is a traversal with $y \in N_{\text{var}}^{@ \mapsto \lambda}$ then so is $t \cdot y \cdot \lambda \bar{\xi} \dots v_{\lambda \bar{\xi}} \cdot v_y$.

$(\text{Value}^{\text{var} \mapsto \lambda})$ If $t \cdot \lambda \bar{\xi} \cdot x \dots v_x$ is a traversal where $x \in N_{\text{var}}$ then so is $t \cdot \lambda \bar{\xi} \cdot x \dots v_x \cdot v_{\lambda \bar{\xi}}$.

In the rest of this chapter we will prove various resulting by induction on the structure of some traversal and by case analysis on the last rule used to form it. For the sake of these proofs, we will use the above-defined reformulation of (Value) in place of its original definition.

2. In the rule (InputValue), although it is not specified, the last node in the traversal $t_1 \cdot x \cdot t_2$ must necessarily belong to $N_{\text{var}} \cup L_{\lambda}$. Indeed, since the pending node x is a variable node, it is of the form


$$\dots \cdot x \cdot \lambda \bar{\eta}_1 \dots v_{\lambda \bar{\eta}_1}^1 \lambda \bar{\eta}_2 \dots v_{\lambda \bar{\eta}_2}^2 \dots \lambda \bar{\eta}_k \dots v_{\lambda \bar{\eta}_k}^k$$

for some nodes $\lambda \bar{\eta}_k$, values $v^k \in \mathcal{D}$ and $k \geq 0$; thus the last occurrence belongs to N_{var} if $k = 0$ and to L_{λ} if $k \geq 1$.


Furthermore, the pending node appears necessarily in the O-view.

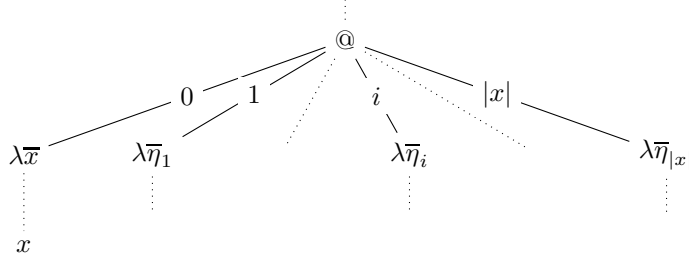
These two observations show that the rule (InputValue) is essentially (InputVar) specialized for value-leaves. The only difference is that (InputVar) allows the visited node to be justified by *any* variable node occurring in the O-view whereas (InputValue) constrains the node to be justified by the pending node (which necessarily occurs in the O-view). This restriction is here to ensure that traversals are well-bracketed.

3. In the rule (Value), it is possible to replace the condition “ $n \in N$ ” by the stronger “ $n \in N \setminus N_{\lambda}^{+@}$ ”. Indeed a later result (Lemma 4.1.6) will show that if n belongs to $N_{\lambda}^{+@}$ then the preceding occurrence m is necessarily an input-variable. Furthermore, another result (Prop. 4.1.1) shows that traversals are well-bracketed, therefore m is necessarily the pending node. Hence the rule (InputValue) can be used in place of (Value) to visit v_m .

The advantage of this alternative formulation is that the traversal rules have a disjoint domains of definition. 

A traversal always starts with the root node and mainly follows the structure of the tree. The exception is the (Var) rule which permits the traversal to jump across the computation tree. The idea is that after visiting a non-input variable node x , a jump can be made to the node corresponding to the subterm that would be substituted for x if all the β -redexes occurring in the term were to be reduced. Let $\lambda \bar{x}$ be x 's binder and suppose x is the i th variable in \bar{x} . The binding node necessarily occurs previously in the traversal (this will be proved in Prop. 4.1.1). Since x is not hereditarily justified by the root, $\lambda \bar{x}$ is not the root of the tree and therefore it is not the first node of the traversal. We do a case analysis on the node preceding $\lambda \bar{x}$:

- If it is an @-node then $\lambda \bar{x}$ is necessarily the first child node of that node and it has i exactly $|\bar{x}|$ siblings: 



In that case, the next step of the traversal is a jump to $\lambda\eta_i$ —the i th child of $@$ —which corresponds to the subterm that would be substituted for x if the β -reduction was performed:

$$t' \cdot @ \cdot \lambda\bar{x} \cdot \dots \cdot x \cdot \lambda\eta_i \cdot \dots \in \text{Trav}(M) .$$

- If it is a variable node y , then the node $\lambda\bar{x}$ was necessarily added to the traversal $t_{\leq y}$ using the (Var) rule. (Indeed, if it was visited using (InputVar) then $\lambda\bar{x}$ would be hereditarily justified by the root. But this is not possible since x_i , bound by $\lambda\bar{x}$, is not an input-variable.) Therefore y is substituted by the term $\kappa(\lambda\bar{x})$ during the evaluation of the term.

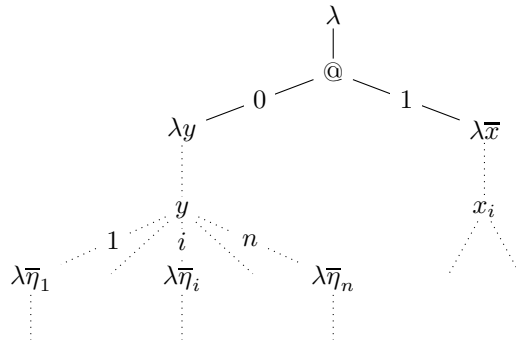
Consequently, during reduction, the variable x will be substituted by the subterm represented by the i th child node of y . Hence the following justified sequence is also a traversal:

$$t' \cdot y \cdot \lambda\bar{x} \cdot \dots \cdot x \cdot \lambda\eta_i \cdot \dots$$

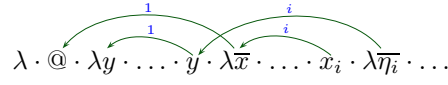
REMARK 4.1.3 Our notions of computation tree and traversal differ slightly from the original definitions by Ong [Ong06a]. In his original setting:

- computation trees contain (uninterpreted first-order) constants. Here we have not accounted for constants but as previously observed, uninterpreted constants can just be regarded as free variables, thus we do not lose any expressivity here.
- constants are restricted to order one at most (terms are used as generators of trees where first-order constants act as tree-node constructors). Here we do not need this restriction: as long as constants are uninterpreted we can regard them as free variables, even at higher-orders.
- one rule ((Sig)) suffices to model the first-order constants. In our setting, however, to account for higher-order variables we have to introduce the more complicated rules (InputValue), (InputVar).
- computation tree do not have value-leaves. These are not necessary to model the pure simply typed lambda calculus. There will be necessary, however, when it comes to model interpreted constants such as those of PCF or IA.

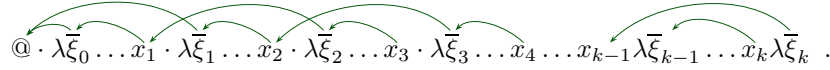
Example 4.1.4. Consider the following computation tree:



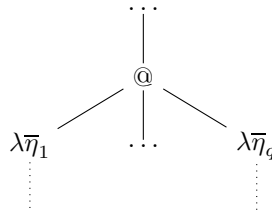
An example of traversal of this tree is:



Lemma 4.1.2. *Take a traversal t ending with an inner node hereditarily justified by an application node $@$. Then if we represent only the nodes appearing in the O -view, the thread of t^ω has the following shape:*



Suppose that the initial node $@$ occurs in the computation as follows:



Let τ_i denote the sub-tree rooted at $\lambda\eta_i$ for $i \in \{1..q\}$. Then for every $j \in \{1..k\}$, x_j and $\lambda\bar{\xi}_j$ must belong to two different subtrees τ_i and $\tau_{i'}$. Furthermore, x_j is hereditarily justified by some occurrence of $\lambda\eta_i$ in t and $\lambda\bar{\xi}_j$ is hereditarily justified by some occurrence of $\lambda\eta_{i'}$ in t (and therefore $\lambda\bar{\xi}_j \in V^{\lambda\eta_{i'} \vdash}$ and $x_j \in V^{\lambda\eta_i \vdash}$).

Proof. The proof is by an easy induction. □

4.1.3.2 Traversal rules for interpreted constants

The framework that we have established up to now aims at providing a computation model of simply typed lambda terms. It is however possible to extend it to other higher-order languages that are based on the simply typed lambda calculus. To achieve, one needs to complete the core traversal rules from Table 4.3 with new rules describing the behaviour of the interpreted constants of the language considered. For instance in the case of PCF, we need to define rules for the interpreted constant **cond** that replicate the behaviour of the conditional operation. (In a forthcoming section of this chapter we will give a complete definition of the constant traversal rules for PCF and IA.)

We mentioned before that uninterpreted constants can be regarded as free variables. In the same way, we can consider interpreted constants as a *generalization* of free variables: for both of them, the “code” describing their computational behaviour is not defined within the scope of the term; it is instead assumed that the environment knows how to interpret them. Free variables, however, have less power than interpreted constants in that when evaluating an applicative term with a free variable in head position, the evaluation of the head variable does not depend on the result of the evaluation of its parameter; whereas for applicative term with an interpreted constant in head position, the evaluation can be driven by the result of the evaluation of its parameter (e.g., the PCF constant **cond** is capable of branching between two of its parameters depending on the result of the evaluation of its first parameter).

Based on these observation, we can define a prototype for constant traversal rules inspired by the definition of the rules (**InputValue**) and (**InputVar**):

Definition 4.1.13 (Constant traversal rule). A **constant traversal** has one of the following two forms:

$$(\Sigma\text{-Value}) \frac{t = t_1 \cdot \alpha \cdot t_2 \in \mathcal{T}rav(M) \quad \alpha \in N_\Sigma \cup N_{\text{var}}^{N_\Sigma \vdash} \quad ?(t)^\omega = \alpha \quad P(t)}{t' = t_1 \cdot \alpha \cdot t_2 \cdot v(t) \in \mathcal{T}rav(M)}$$

or

$$(\Sigma)/(\Sigma\text{-Var}) \frac{t \in \mathcal{Trav}(M) \quad t^\omega \in N_\Sigma \cup N^{N_{\Sigma^\perp}} \cup L_\lambda \quad P(t)}{t \cdot n(t) \in \mathcal{Trav}(M)}$$

where:

- $P(t)$ is a predicate expressing some condition on t ;
- $v(t)$ is a value-leaf of the node α that is determined by the traversal t ;
- $n(t)$ is a lambda-node determined by t , and its link, also determined by t , points to some occurrence of its parent node in $\perp t \perp$.

Clearly, such rules preserve well-bracketing, alternation and visibility.

REMARK 4.1.4 The extra power of the constant rules over the input-variable rules (**InputValue**) and (**InputVar**) comes from their ability to base their choice of the next node to visit on the shape of the traversal t .

From now on, to make our argument as general as possible, we consider a simply typed lambda-calculus language extended with higher-order interpreted constants for which some constant traversal rules have been defined (in the sense of Def. 4.1.13). Furthermore, we complete the set of rules with the following additional copy-cat rule:

$$(\text{Value}^{\Sigma \mapsto \lambda}) \quad t \cdot \lambda \bar{\xi} \cdot \overset{v}{\curvearrowright} c \dots v_c \in \mathcal{Trav}(M) \wedge c \in \Sigma \implies t \cdot \lambda \bar{\xi} \cdot \overset{v}{\curvearrowright} c \dots v_c \cdot v_{\lambda \bar{\xi}} \in \mathcal{Trav}(M) .$$

We now introduce a special kind of constant traversal rules:

Definition 4.1.14. A constant traversal rule is **well-behaved** if for every traversal $t \cdot \alpha \cdot u \cdot n$ formed with the rule we have $?(u) = \epsilon$.

The rule (Σ -Value), for instance, is clearly well-behaved (because of well-bracketing of traversals), whereas $(\Sigma)/(\Sigma\text{-Var})$ is not well-behaved in general since $n(t)$ does not necessarily points to the pending node in t .

Lemma 4.1.3. *If Σ -constants have order 1 at most, then constant rules are necessarily all well-behaved.*

Proof. In the computation tree, an order-1 constant hereditarily enables only its immediate children (which are all dummy lambda nodes λ). Hence a traversal formed with the rule $(\Sigma)/(\Sigma\text{-Var})$ is of the form:

$$t = \dots \cdot \alpha \cdot u \cdot \lambda$$

where α appears in $\perp t \perp$.

If $u = \epsilon$ then the result trivially holds. Otherwise, u 's first node has necessarily been visited with the rule $(\Sigma)/(\Sigma\text{-Var})$ thus u 's first node is a dummy lambda node λ' pointing to α . Since α occurs in $\perp t \perp$ and since the node λ' enables only its value-leaf in the computation tree, t must be of the following shape:

$$t = \dots \cdot \alpha \cdot \underbrace{\lambda' \dots v_{\lambda'} \dots \lambda}_u$$

for some value leaf $v_{\lambda'}$ of λ' .

Again, the node following $v_{\lambda'}$ must be a dummy lambda node pointing to α . By iterating the same argument we obtain that the segment u is a repetition of segments of the form $\lambda' \dots v_{\lambda'}$. Hence $?(u) = \epsilon$. \square

4.1.3.3 Property of traversals

Proposition 4.1.1. *Let t be a traversal. Then:*

- (i) t is a well-defined and well-bracketed justified sequence;
- (ii) t is a well-defined justified sequence verifying alternation, P-visibility and O-visibility;
- (iii) If $t^\omega \notin L_\lambda$ (i.e., t 's last occurrence is not a lambda value-leaf), then $\ulcorner t \urcorner$ is the path in the computation tree going from the root to the node t^ω .

Proof. This is the counterpart of another result proved by Ong in the paper where he introduces the theory of traversals [Ong06b, proposition 6]. The original proof—an induction on the traversal rules—can be adapted to take into account the constant rules and the presence of value-leaves in the traversal. We detail the case (Lam) only. We need to show that n 's binder occurs only once in the P-view at that point. By the induction hypothesis (iii) we have that $\ulcorner t \cdot \lambda \bar{\xi} \urcorner$ is a path in the computation tree from the root to $\lambda \bar{\xi}$. But n 's binder occurs only once in this path, therefore the traversal $t \cdot \lambda \bar{\xi} \cdot n$ is well-defined and satisfies P-visibility. Thus (i) and (ii) are verified. Furthermore n is a child of $\lambda \bar{\xi}$ therefore (iii) also holds. \square

Lemma 4.1.4. *If $t \cdot n$ is a traversal with $n \in N_{\text{var}} \cup N_\Sigma \cup N_\textcircled{\text{a}}$ then t^ω is a lambda node and is n 's parent in $\tau(M)$. (Thus t^ω is not a leaf: $n \notin L$).*

Proof. By inspecting the traversal rules, we observe that (Lam) is the only rule which can visit a node in $N_{\text{var}} \cup N_\Sigma \cup N_\textcircled{\text{a}}$. Hence t^ω must be n 's parent in $\tau(M)$. \square

Lemma 4.1.5. *Suppose that M is β -normal and let t be a traversal of $\tau(M)$. For any node n occurring in t , $\textcircled{*}$ does not hereditarily enable n if and only if n is hereditarily enabled by some node in N_Σ . Formally:*

$$n \notin N^{\textcircled{*}\vdash} \iff n \in N^{N_\Sigma\vdash}.$$

Proof. In a computation tree, the only nodes that do not have justification pointer are: the root $\textcircled{*}$, $\textcircled{\text{a}}$ -nodes and Σ -constant nodes. But since M is in β -normal form, there is no $\textcircled{\text{a}}$ -node in the computation tree. Hence nodes are either hereditarily enabled by $\textcircled{*}$ or hereditarily enabled by some node in N_Σ . Moreover $\textcircled{*}$ is not in N_Σ therefore the “or” is exclusive : a node cannot be both hereditarily enabled by $\textcircled{*}$ and by some node in N_Σ . \square

Lemma 4.1.6 (The O-view is contained in a single thread). *Let $t \in \text{Trav}(M)$.*

- (a) *If $t = \dots \cdot m \cdot n$ where $m \in N_{\text{var}} \cup N_\Sigma \cup N_\textcircled{\text{a}} \cup L_\lambda$ and $n \in N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\textcircled{\text{a}}$ then m and n are in the same thread in t : they are her. just. by the same initial occurrence (which is either $\tau(M)$'s root, a Σ -constant or an $\textcircled{\text{a}}$ -node);*
- (b) *All the nodes in $\ulcorner t \urcorner$ belong to the same thread.*

Proof. Clearly (b) follows immediately from (a) due to the way the O-view is computed. We show (a) by induction on the last traversal rule used to form t . The results trivially hold for the base cases (Empty) and (Root). Step case: Take $t = t' \cdot n$. If $n \in N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\textcircled{\text{a}}$ then we do not need to show (a). Otherwise $n \in N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\textcircled{\text{a}}$. By O-visibility, n points in $\ulcorner t' \urcorner$, thus by the I.H., it must belong to the same thread as all the nodes in $\ulcorner t' \urcorner$ and in particular to the thread of t'^ω . Therefore both (i) and (ii) hold. \square

4.1.3.4 Traversal reduction

A traversal ending with an input-variable can be extended in a non-deterministic way. Occurrences of these nodes correspond to point of the computation at which the term interacts with its context. In contrast, if a traversal ends with a node hereditarily enabled by an $\textcircled{\text{a}}$ -node or by a constant nodes then the next visited node is uniquely determined. We can therefore think of them as nodes “internal” to the computation: their semantics is predefined and cannot be altered by the context

in which the term appears. Hence, if we want to extract the essence of the computation from a traversal, a natural way to proceed is to keep only occurrences of nodes that are hereditarily enabled by the root:

Definition 4.1.15. The *reduction of a traversal* t , written $t \upharpoonright \circledast$, is defined as the subsequence of t consisting of the occurrences of nodes that are hereditarily enabled by the root \circledast of the computation tree.

Example 4.1.5. The reduction of the traversal given in example 4.1.3 is:

$$t \upharpoonright \lambda fz = \lambda fz \cdot f \cdot \lambda \cdot z .$$

REMARK 4.1.5

- The root occurs at most once in a traversal, therefore if t is a non-empty traversal then the reduction of t is equal to $t \upharpoonright r$ where r denotes the only occurrence of \circledast in t .
- Since $@$ -nodes and Σ -constants do not have pointers, the reduction of traversal contains only nodes in $V_\lambda \cup V_{\text{var}}$.

We define $\mathcal{Trav}(M)^{\upharpoonright \circledast}$ as the set of reductions of traversals of M :

$$\mathcal{Trav}(M)^{\upharpoonright \circledast} = \{t \upharpoonright r : t \in \mathcal{Trav}(M) \text{ and } r \text{ is the only occurrence of } \circledast \text{ in } t\} .$$

4.1.3.5 Removing $@$ -nodes and Σ -nodes from traversals

When defining computation trees, it was necessary to introduce application nodes (labelled $@$) in order to connect the operator and the operand of an application. The presence of $@$ -nodes has also another advantage: it ensures that the lambda-nodes are all at even level in the computation tree, and thus a traversal respects a certain form of alternation between lambda nodes and non-lambda nodes. Application nodes are however redundant in the sense that they do not play any role in the computation of the term. In fact it will be necessary to filter them out in order to establish the correspondence with the interaction game semantics.

Definition 4.1.16 ($@$ -free traversal). Let t be a traversal of $\tau(M)$. We write $t - @$ for the sequence of nodes-with-pointers obtained by

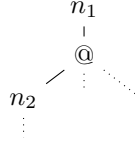
- removing from t all $@$ -nodes and value-leaves of some $@$ -node;
- replacing any link pointing to an $@$ -node by a link pointing to the immediate predecessor of $@$ in t .

Suppose $u = t - @$ is a sequence of nodes obtained by applying the previously defined transformation on the traversal t , then t can be partially recovered from u by reinserting the $@$ -nodes as follows. For each $@$ -node in the computation tree with parent node denoted by p , we perform the following operations:

1. replace every occurrence of the pattern $p \cdot n$ for some λ -node n , by $p \cdot @ \cdot n$;
2. replace any link in u starting from a λ -node and pointing to p by a link pointing to the inserted $@$ -node;
3. for each occurrence in u of a value-leaf v_p pointing to p , insert the value-leaf $v_{@}$ immediately before v_p and make it point to the immediately successor of p (which is precisely the $@$ -node inserted in step 1.).

We write $u + @$ for this second transformation.

These transformations are well-defined because in a traversal, an $@$ -node is always immediately preceded by its parent node n_1 , and immediately followed by its first child n_2 :



Example 4.1.6. Let f be a Σ -constant and $t = \lambda \bar{\xi} \cdot @ \cdot \lambda x \cdot f \cdot \lambda \cdot x$. Then

$$t - @ = \lambda \bar{\xi} \cdot \lambda x \cdot f \cdot \lambda \cdot x .$$

Example 4.1.7. Let t be the traversal given in example 4.1.3, we have:

$$t - @ = \lambda f z \cdot \lambda uv \cdot u \cdot \lambda y \cdot f \cdot \lambda \cdot y \cdot \lambda \cdot v \cdot \lambda \cdot z .$$

We also want to remove Σ -nodes from the traversals. To that end we define the operation $-\Sigma$ and $+\Sigma$ in the exact same way as $-@$ and $+@$. Again these transformations are well-defined since in a traversal, a Σ -node f is always immediately preceded by its parent node p , and a value-node v_p is always immediately preceded by a value-node v_f .

Note that the operations $-@$ and $-\Sigma$ are commutative: $(t - @) - \Sigma = (t - \Sigma) - @$.

Lemma 4.1.7. For any traversal t in $\mathcal{Trav}(M)$:

$$\begin{aligned} (t - @) + @ &= \begin{cases} t, & \text{if } t^\omega \notin V_{@} ; \\ \text{ip } t, & \text{if } t^\omega \in V_{@} ; \end{cases} \\ (t - \Sigma) + \Sigma &= \begin{cases} t, & \text{if } t^\omega \notin V_{\Sigma} ; \\ \text{ip } t, & \text{if } t^\omega \in V_{\Sigma} . \end{cases} \end{aligned}$$

Proof. The result follows immediately from the definition of the operation $-@$ and $+@$ (resp. $-\Sigma$ and $+\Sigma$). \square

REMARK 4.1.6 Sequences of the form $t - @$ (resp. $t - \Sigma$) are not, strictly speaking, proper justified sequences of nodes since after removing $@$ -nodes, all the prime λ -nodes become justified by their parent's parent which are also λ -nodes! Moreover, these sequences do not respect alternation since two λ -nodes may become adjacent after removing a $@$ -node.

We write t^* to denote the sequence obtained from t by removing all the $@$ -nodes as well as the constant nodes together with their associated value-leaves:

$$t^* \stackrel{\text{def}}{=} t - @ - \Sigma .$$

Example 4.1.8. Let f be a Σ -constant. We have

$$\left(\lambda \bar{\xi} \cdot @ \cdot \lambda x \cdot f \cdot \lambda \cdot x \right)^* = \lambda \bar{\xi} \cdot \lambda x \cdot \lambda \cdot x .$$

We introduce the set

$$\mathcal{Trav}(M)^* = \{t^* \mid t \in \mathcal{Trav}(M)\} .$$

REMARK 4.1.7 If M is a β -normal term and if there are no Σ -constant (as in the simply typed lambda calculus) then $\tau(M)$ does not contain any $@$ -node or Σ -node therefore all nodes are hereditarily enabled by \otimes and we have $\mathcal{Trav}(M) = \mathcal{Trav}(M)^{\dagger \otimes} = \mathcal{Trav}(M)^*$.

We extend the notion of P-view to sequences of the form t^* (the definition is the same as for traversals). It is easy to check that if t^ω is not in $L_{@} \cup L_{\Sigma}$ then the P-view of t^* is obtained from $\ulcorner t^\omega \urcorner$ by keeping only the non $@/\Sigma$ -nodes:

$$\ulcorner t^{*\omega} \urcorner = \ulcorner t^\omega \urcorner \setminus (V_{@} \cup V_{\Sigma}) . \quad (4.1)$$

We define a projection operation for sequences of the form t^* as follows:

Definition 4.1.17. Let t be a traversal such that $t^\omega \notin L_{@} \cup L_{\Sigma}$ and r_0 be an occurrence of some lambda-node n . Then the projection $t^* \upharpoonright V^{(n)}$ is defined as the subsequence of t^* constituted of nodes of $V^{(n)}$ only. If a variable node loses its pointer in $t^* \upharpoonright V^{(n)}$ then its justifier is reassigned to the only occurrence of n in $\ulcorner t^{*\omega} \urcorner$.

Note that this operation is well-defined. Indeed if a variable x loses its pointer in $t^* \upharpoonright V^{(n)}$ then it means that x is free in $M^{(n)}$. But then n must occur in the path to the root \otimes which is precisely $\ulcorner t_{\leq x}^\omega \urcorner$. Thus by Eq. 4.1, n must occur in $\ulcorner t_{\leq x}^{*\omega} \urcorner$.

4.1.3.6 Subterm projection (with respect to a reference node occurrence)

We fix a reference node-occurrence n_0 in a traversal t . The **subterm projection** $t \upharpoonright^+ n_0$ is defined to be the subsequence of t consisting of the occurrences whose P-view at that point contain the node n_0 . Formally:

Definition 4.1.18. Let $t \in \text{Trav}(M)$ and n_0 be an occurrence in t call the *reference node*. The subsequence $t \upharpoonright^+ n_0$ of t is defined inductively on t as follows:

- $(t \cdot n_0) \upharpoonright^+ n_0 = n_0$;
- If $n \in N_{\lambda} \cup L_{\text{var}} \cup L_{\Sigma} \cup L_{@}$ and $n \neq n_0$ then

$$(t \cdot n) \upharpoonright^+ n_0 = \begin{cases} (t \upharpoonright^+ n_0) \cdot n, & \text{if } n\text{'s justifier appears in } t \upharpoonright^+ n_0 ; \\ t \upharpoonright^+ n_0, & \text{otherwise;} \end{cases}$$

- If $n \in N_{\text{var}} \cup N_{\Sigma} \cup N_{@} \cup L_{\lambda}$ and $n \neq n_0$ then

$$(t \cdot n) \upharpoonright^+ n_0 = \begin{cases} (t \upharpoonright^+ n_0) \cdot n, & \text{if } t^\omega\text{'s appears in } t \upharpoonright^+ n_0 ; \\ t \upharpoonright^+ n_0, & \text{otherwise;} \end{cases}$$

where in the first subcase, if n loses its justifier in $t \upharpoonright^+ n_0$ then it is reassigned to r_0 .

We call this transformation the subterm projection *with respect to a reference node occurrence* because it keeps only nodes that appear in the sub-tree rooted at this node. If n_0 is an occurrence of a lambda node $n \in N_{\lambda}$ then we will call $t \upharpoonright^+ n_0$ a **sub-traversal of the computation tree** $\tau(M)$. This appellation is suggestive of the forthcoming Proposition 4.1.5 stating that $t \upharpoonright^+ n_0$ is a traversal of the sub-computation tree of $\tau(M)$ rooted at n .

REMARK 4.1.8 There is an alternative way to define $t \upharpoonright^+ r_0$: For any traversal t we write t^+ to denote the sequence-with-pointers obtained from t by adding pointers as follows: for every occurrence of a $@$ or Σ node m in t we add a pointer going from m to its predecessor in t (which is necessarily an occurrence of its parent node). Further, for every variable node x we add auxiliary pointers going to each lambda nodes occurring in the P-view at that point after x 's binder. Conversely, for any sequence-with-pointers u we define u^- as the sequence obtained from u by removing the links associated to $@$ and Σ -nodes and where for each occurrence of a variable node, only the “longest” link is preserved. (The length of a link being defined as the distance between the source and the target occurrence.) Clearly the operation $-$ is the inverse of $+$: for any traversal t we have $t = (t^+)^-$. Then it can be easily shown that the sequence $t \upharpoonright^+ n$ is precisely the subsequence of t constituted of nodes hereditarily justified by n *with respect to the justification pointers of t^+* :

$$t \upharpoonright^+ n = (t^+ \upharpoonright n)^- .$$

(Note that since the operation $-$ changes the justification pointers, the hereditary justification relation in a traversal t is different from the hereditary justification relation in t^+ and therefore we have $(t \upharpoonright n)^+ \subseteq t^+ \upharpoonright n$ but $(t \upharpoonright n)^+ \neq t^+ \upharpoonright n$.) End of remark.

The following lemmas follow directly from the definition of $t \upharpoonright^+ r_0$:

Lemma 4.1.8. *Let t be a traversal and r_0 be an occurrence of a lambda node r' in t .*

- (a) *Suppose that $t = \dots \overleftarrow{m} \dots n$ with $n \in N_\lambda \cup L_\oplus \cup L_\Sigma \cup L_{\text{var}}$ and $n \neq r_0$. Then n appears in $t \upharpoonright^+ r_0$ if and only if m appears in $t \upharpoonright^+ r_0$.*
- (b) *Suppose that $t = \dots n$ where $n \in N_{\text{var}} \cup N_\oplus \cup N_\Sigma \cup L_\lambda$. Then n appears in $t \upharpoonright^+ r_0$ if and only if the last lambda node in $\ulcorner t \urcorner$ does.*
- (c) *Suppose that $t = \dots \overleftarrow{m} \dots v_m$ with $v_m \in L = L_\lambda \cup L_\oplus \cup L_\Sigma \cup L_{\text{var}}$. Then v_m appears in $t \upharpoonright^+ r_0$ if and only if m does.*

Proof. (a) holds by definition of $t \upharpoonright^+ r_0$. (b) is due to the fact that $t \upharpoonright^+ r_0$ is defined by induction with inductive steps following the structure of those used in the definition of the traversal P-view. The formal proof is by a trivial induction on t . (c) If $v_m \in L_\oplus \cup L_\Sigma \cup L_{\text{var}}$ then it falls back to (a). Otherwise $v_m \in L_\lambda$ and by (b), v_m appears in $t \upharpoonright^+ r_0$ if and only if the last lambda node in $\ulcorner t \urcorner$ does. But the last node in $\ulcorner t \urcorner$ is necessarily m (since v_m is necessarily visited with a copy-cat rule). \square

Lemma 4.1.9. *Let $t \in \text{Trav}(M)$ and r_0 be the occurrence in t of a λ -node. We have:*

$$?(t \upharpoonright^+ r_0) = ?(t) \upharpoonright^+ r_0 .$$

Proof. Take a prefix u of t ending with a value-leaf v_n of an occurrence n . By Lemma 4.1.8(c), the operation $_ \upharpoonright^+ r_0$ removes v_n from t if and only if also removes n . \square

Lemma 4.1.10. *For any non-empty traversal t we have $t^\star \upharpoonright V^{\oplus\perp} = t \upharpoonright r$ where r denotes the only occurrence of the root in t .*

Proof. The result follows from the following two observations: 1. Node occurrences that are not her. just. by r in t may become her. just. by r in t^\star , however these nodes are not hereditarily enabled by $\tau(M)$'s root and thus are not in $V^{\oplus\perp}$. 2. Conversely, all the node occurrences that are hereditarily justified by r in t are necessarily hereditarily enabled by the root and therefore appear in $t^\star \upharpoonright V^{\oplus\perp}$. \square

4.1.3.7 O-view and P-view of the subterm projection

P-view projection

Lemma 4.1.11 (P-view Projection for traversals). *Let t be a traversal and r_0 be an occurrence in t of a lambda node $r' \in N_\lambda$. Then:*

- (i) *If t^ω appears in $t \upharpoonright^+ r_0$ then*
 - a. *r_0 appears in $\ulcorner t \urcorner$, all the nodes occurring after r_0 in $\ulcorner t \urcorner$ appear in $t \upharpoonright^+ r_0$ and all the nodes occurring before r_0 in $\ulcorner t \urcorner$ do not appear in $t \upharpoonright^+ r_0$;*
 - b. $\ulcorner t \urcorner \upharpoonright^+ r_0 \ulcorner M^{(r')} \urcorner = \ulcorner t \urcorner \ulcorner M \urcorner_{\geq r_0} = r_0 \dots$;
 - c. *If t^ω also appears in $t \upharpoonright^+ r_1$ for some occurrence r_1 r' then $r_0 = r_1$;*
 - d. *if $t = \dots \overleftarrow{m} \dots n$ and m does not appear in $t \upharpoonright^+ r_0$ then r_0 occurs after m in t and m is a free variable node in the sub-computation tree $\tau(M^{(r')})$.*
- (ii) *Suppose $t = \dots r_0 \dots \overleftarrow{m} \dots n$. The node n appears in $t \upharpoonright^+ r_0$ if and only if m does.*

Proof. (i) a. and b. The projection operation $_ \upharpoonright^+ r_0$ is defined by an induction whose inductive steps correspond precisely to those used in the P-view computation. Thus a trivial induction shows that both a. and b. hold.

c. By a., both r_0 and r_1 appears in the P-view. But the P-view is the path from t^ω to the root, hence it cannot contain two different occurrences of the same node r' .

d. Since t^ω appears in $t \upharpoonright^+ r_0$ and its justifier m is not in $t \upharpoonright^+ r_0$, by (i)a. the justifier m necessarily precedes r_0 in t , and by Lemma 4.1.8, n is necessarily a variable node. Thus m occurs before r_0 in the P-view $\lceil t \rceil$. In other words, r_0 lies in the path from n to its binder m . Consequently, n is a free variable node in $\tau(M^{(r')})$.

(ii) The case $n \notin N_{\text{var}}$ is handled by Lemma 4.1.8(a) and (c). Suppose that $n \in N_{\text{var}}$. If n appears in $t \upharpoonright^+ r_0$ then by (i) all the nodes occurring in $\lceil t \rceil$ up to r_0 appear in $t \upharpoonright^+ r_0$. By P-visibility, m appears in $\lceil t \rceil$ and since r_0 precedes it by assumption, m also appears in $t \upharpoonright^+ r_0$.

If m appears in $t \upharpoonright^+ r_0$ then clearly, by definition of $t \upharpoonright^+ r_0$, since m appears in the P-view at x , x must also appears in $t \upharpoonright^+ r_0$. \square

Lemma 4.1.12. *Let $t \in \text{Trav}(M)$ such that $t^\omega \notin L_\lambda$. Let r' be some lambda node in N_λ .*

The node t^ω belongs to the subtree of $\tau(M)$ rooted at r' (i.e., $t^\omega \in V^{(r')}$) if and only if t^ω appears in $t \upharpoonright^+ r_0$ for some occurrence r_0 of r' in t .

Proof. Only if part: Since t 's last move is not a lambda leaf, by Proposition 4.1.1, the P-view $\lceil t \rceil$ gives the path to the root \otimes hence since t^ω belongs to the subtree of $\tau(M)$ rooted at r' , $\lceil t \rceil$ must contain (exactly) one occurrence r_0 of r' . But then by definition of $t \upharpoonright^+ r_0$, all the nodes following r_0 occurring in the P-view must also belong to $t \upharpoonright^+ r_0$. In particular, t^ω appears in $t \upharpoonright^+ r_0$.

If part: By Lemma 4.1.11(i), r_0 must occur in $\lceil t \rceil$ and therefore r_0 lies in the path from t^ω to the root \otimes of the computation tree $\tau(M)$. Consequently, t^ω necessarily belongs to the subtree of $\tau(M)$ rooted at r' . \square

Lemma 4.1.13. *Let t be a traversal such that $t^\omega \notin V_\otimes \cup V_\Sigma$ and r_0 be an occurrence of a lambda node r' in t . Then an occurrence n is her. just. by n_0 in $t^* \upharpoonright V^{(r')}$ if and only if n appears in $t \upharpoonright^+ r_0$.*

Proof. We suppose that n is the last node in t and we proceed by induction on t . If $n = r_0$ or if r_0 does not occur in t then the result holds trivially. Suppose that r_0 occurs in $\text{ip}(t)$. Let m be n 's justifier in t . The case $n \in L_\otimes \cup L_\Sigma \cup N_\otimes \cup N_\Sigma$ is excluded by the hypothesis.

Suppose $n \in L_\lambda \cup L_{\text{var}} \cup N_\lambda$ then

$$\begin{aligned} n \text{ appears in } t \upharpoonright^+ r_0 &\iff m \text{ appears in } t \upharpoonright^+ r_0 && \text{by Lemma 4.1.8(a)} \\ &\iff m \text{ her. just. by } n_0 \text{ in } t^* \upharpoonright V^{(r')} && \text{by I.H. on } t_{\leq m} \\ &\iff n \text{ her. just. by } n_0 \text{ in } t^* \upharpoonright V^{(r')} && \text{since } m \text{ is } n\text{'s parent in } \tau(M^{(r')}). \end{aligned}$$

Suppose that $n \in N_{\text{var}}$.

$$\begin{aligned} n \text{ appears in } t \upharpoonright^+ r_0 &\iff r_0 \text{ appears in } \lceil t \rceil && \text{(by Lemma 4.1.12 and 4.1.11(i))} \\ &\iff \begin{cases} r_0 \text{ precedes } m \text{ in } \lceil t \rceil, \text{ and thus } n \text{ is a bound variable in } M^{(r')} \\ \text{or } r_0 \text{ appears strictly after } m \text{ in } \lceil t \rceil \text{ and } n \text{ is free in } M^{(r')} \end{cases} \\ &\iff \begin{cases} m \text{ appears in } t \upharpoonright^+ r_0 && \text{(by Lemma 4.1.11(i))} \\ \text{or } n \text{ points to } r_0 \text{ in } t^* \upharpoonright V^{(r')} && \text{(by def. of } _ \upharpoonright V^{(r')}) \end{cases} \\ &\iff \begin{cases} m \text{ her. just. by } n_0 \text{ in } t^* \upharpoonright V^{(r')} && \text{(by I.H. on } t_{\leq m}) \\ \text{or } n \text{ points to } r_0 \text{ in } t^* \upharpoonright V^{(r')}. \end{cases} \\ &\iff \begin{cases} n \text{ her. just. by } n_0 \text{ in } t^* \upharpoonright V^{(r')} && (n \text{ is in } V^{(r')} \text{ iff its binder } m \text{ is}) \\ \text{or } n \text{ points to } r_0 \text{ in } t^* \upharpoonright V^{(r')} \end{cases} \\ &\iff n \text{ is her. just. by } n_0 \text{ in } t^* \upharpoonright V^{(r')} && \square \end{aligned}$$

Lemma 4.1.14. *Take a traversal t . Let r' be a node in N_λ and r_0 an occurrence of r' in t . Suppose that t^ω appears in $t \upharpoonright^+ r_0$ and that the thread of t^ω is initiated by $\alpha \in N_\oplus \cup N_\Sigma$.*

- (i) *If r_0 precedes α in t then all the nodes occurring in the thread appear in $t \upharpoonright^+ r_0$.*
- (ii) *If α precedes r_0 in t then t^ω is hereditarily enabled by r' in $\tau(M^{(r')})$.*

Proof. (i) By definition of a thread, the nodes occurring in the thread are all hereditarily justified by α . Since r_0 precedes α and t^ω appears in $t \upharpoonright^+ r_0$, by Lemma 4.1.11(ii) all the nodes in the thread must also appear in $t \upharpoonright^+ r_0$.

(ii) Let q be the first node in t that hereditarily justifies t^ω in t and that appears in $t \upharpoonright^+ r_0$.

If $q \in N_\lambda$ then necessarily $q = r_0$. Otherwise by definition of $\upharpoonright^+ r_0$, q 's justifier also appears in $t \upharpoonright^+ r_0$ which contradicts the definition of q . Hence the result holds trivially.

If $q \in N_\oplus \cup N_\Sigma$ then necessarily $q = \alpha$, since links always point inside the current thread and since a thread contains by definition only one node in $N_\oplus \cup N_\Sigma$. But α precedes r_0 therefore α cannot be her. just. by r_0 hence this case is not possible.

If $q \in N_{\text{var}}$ then by Lemma 4.1.11(i.d), q is a free variable in $\tau(M^{(r')})$ and therefore it is enabled by r' in $\tau(M^{(r')})$. Hence since t^ω is her. just. by r_0 , it must be hereditarily enabled by r' in $\tau(M^{(r')})$. \square

O-view projection In this paragraph we will spend some time proving the following Proposition:

Proposition 4.1.2 (O-view projection for traversals). *Let t be a traversal of $\text{Trav}(M)$ such that its last node appears in $t \upharpoonright^+ r_0$ for some occurrence r_0 in t of a lambda node r' in N_λ . Then $\sqsubseteq t \upharpoonright_M \upharpoonright^+ r_0 \sqsubseteq \sqsubseteq t \upharpoonright^+ r_0 \upharpoonright_{M^{(r')}}$.*

Unfortunately, this result is relatively hard to prove. Note, however, that it bears resemblance with another non trivial result of game semantics found in the original paper by Hyland and Ong on full abstraction of PCF [HO00]:

Proposition 4.1.3 (P-view projection in game semantics). [HO00, Prop.4.3] *Let s be a legal position of a game $A \rightarrow B$. If s^ω is in B then $\ulcorner s \urcorner^{A \rightarrow B} \upharpoonright B \sqsubseteq \ulcorner s \urcorner \upharpoonright B^\omega$.*

Instead of redoing a similar proof for our slightly different setting, we just show how to recast this result in it. The proof of the previous proposition relies on several properties of a legal position s [HO00]:

- (w1) Initial question to start: there first move played in s is an initial move and there is no other occurrence of initial moves in the rest of s ;
- (w2) Alternation: P-moves and O-moves alternate in s ;
- (w3) Explicit justification: *every* move except the first has a pointer to a preceding move,
- (w4) Well-bracketing: the pending question is answered first;
- (w5) Visibility: s satisfies P-visibility and O-visibility.

Also, further assumptions are made on the legal positions of the game $A \rightarrow B$:

- (w6) For any occurrence n in the position, $n \in A \iff n \notin B$;
- (w7) Switching condition: the Proponent is the only player who can switch from game A to B or from B to A .
- (w8) Justification in $A \rightarrow B$: Suppose m justifies n in s . Then
 - $n \in B$ implies $m \in B$;
 - if n is a non-initial move in A then $n \in A$;

- if n is an initial move in A the $n \in B$.

Most of these requirements coincide with properties that we have already shown for traversals. However traversals do not strictly satisfy explicit justification since there are some nodes in the traversal that do not have justification pointers: the $@$ -nodes and Σ -nodes. To overcome this problem, well, we just add justification pointers to those nodes!

Take a justified sequence of nodes t . We define $\text{ext}(t)$, the *extension of t* , to be the sequence of nodes-with-pointers obtained from $\diamond \cdot t$ (where \diamond is a dummy node) by adding justification pointers going from occurrences of the root \otimes , $@$ -nodes and Σ -nodes to their immediate predecessor in t .

Example 4.1.9. Let $f \in \Sigma$. We have $\text{ext}(\lambda \bar{x} \cdot @ \cdot \lambda x \cdot f \cdot \lambda \cdot x) = \diamond \cdot \lambda \bar{x} \cdot @ \cdot \lambda x \cdot f \cdot \lambda \cdot x$.

It is an immediate fact that for any two justified sequences t_1 and t_2 we have:

$$\text{ext}(t_1) \sqsubseteq \text{ext}(t_2) \iff t_1 \sqsubseteq t_2 \quad (4.2)$$

and for any justified sequence t :

$$\text{ext}(t) \uparrow^+ r_0 = \text{ext}(t \uparrow^+ r_0) . \quad (4.3)$$

Since a traversal extension $\text{ext}(t)$ may contain $@/\Sigma$ -nodes with pointers, they are not proper justified sequences of nodes as defined in Def. 4.1.6. Nevertheless, the basic transformations that we have defined for justified sequences, such as hereditary projection, P-view and O-view, apply naturally to traversal extensions (without any modification in their definition). The views of a traversal extension can be expressed in term of the traversal's views as follows:

$$\perp \text{ext}(t) \perp = \perp t \perp \quad (4.4)$$

$$\ulcorner \text{ext}(t) \urcorner = \begin{cases} \epsilon, & \text{if } t = \epsilon ; \\ \diamond \cdot \text{ext}(\ulcorner t \urcorner), & \text{otherwise.} \end{cases} \quad (4.5)$$

The transformations $\ulcorner \cdot \urcorner$ and $\perp \cdot \perp$, however, do not convey the appropriate notion of view for extended traversals. We define an alternative notion of view more appropriate to traversal extensions, called O-e-view and P-e-view, as follows:

Definition 4.1.19. The O-e-view of a traversal extension $\text{ext}(t)$, written, $\perp \text{ext}(t) \perp_e$ is defined as

$$\perp \text{ext}(t) \perp_e \stackrel{\text{def}}{=} \ulcorner \text{ext}(t) \urcorner .$$


The P-e-view of $\text{ext}(t)$, written, $\perp \text{ext}(t) \perp_e$ is defined by induction:

$$\begin{aligned} \ulcorner \epsilon \urcorner^e &= \epsilon \\ \ulcorner u \cdot n \urcorner^e &= \ulcorner u \urcorner^e \cdot n \quad \text{for } n \in L_{\text{var}} \cup L_{\Sigma} \cup L_{@} \cup N_{\lambda} ; \\ \ulcorner u \cdot \overleftarrow{m \dots n} \urcorner^e &= \ulcorner u \urcorner^e \cdot \overleftarrow{m \cdot n} \quad \text{for } n \in N_{\text{var}} \cup L_{\lambda} \cup N_{@} \cup N_{\Sigma} . \end{aligned}$$

By inserting a dummy node \diamond at the beginning of the traversal, we change the parity of the alternation between nodes in $N_{\text{var}} \cup L_{\lambda} \cup N_{@} \cup N_{\Sigma}$ and $N_{\lambda} \cup L_{\text{var}} \cup L_{\Sigma} \cup L_{@}$. Thus the role of O and P is interchanged for traversal extensions. This explains why the O-e-view is calculated from the P-view.

For the P-e-view, the definition is almost the same as the traversal O-view $\perp \cdot \perp$ except that the computation does not stop when reaching a node in $N_{@} \cup N_{\Sigma}$ —this is sometimes referred as the *long O-view* [Har05]. (The O-view contains only one thread whereas the long-O-view may contain several; the O-view is a suffix of the long O-view.) This is possible because occurrences of nodes from $N_{@} \cup N_{\Sigma}$ in a traversal extension all have a justification pointer. The O-view of t is a suffix of its P-e-view:

$$\ulcorner t \urcorner^e = w \cdot \perp t \perp \quad \text{for some sequence } w. \quad (4.6)$$

We are now fully equipped to establish an analogy between the traversal extension setting and the game semantic setting. The reason why we make this analogy is purely to reuse the proof of Proposition 4.1.3 [HO00]. The reader must not confuse this with a different correspondence, that we will establish in a forthcoming section  between plays of game semantics and traversals of the computation tree. (In particular the coloring of nodes used here in term of P-move/O-move is the opposite of the one used in the Correspondence Theorem.) The following analogy is made:

Traversal setting	Game-semantic setting
Extended traversal $\text{ext}(t)$	Play s
Nodes in $n \in N_{\text{var}} \cup L_{\lambda} \cup N_{@} \cup N_{\Sigma} \cup \{\diamond\}$	O-moves \bullet
Nodes in $n \in N_{\lambda} \cup L_{\text{var}} \cup L_{\Sigma} \cup L_{@}$	P-moves \circ
P-view $\lceil \text{ext}(t) \rceil^e$	P-view $\lceil s \rceil$
O-view $\lfloor \text{ext}(t) \rfloor_e$	O-view $\lfloor s \rfloor$
Occurrence n appearing in $t \upharpoonright^+ r_0$	Occurrence $n \in B$
Occurrence n not appearing in $t \upharpoonright^+ r_0$	Occurrence $n \in A$
No notion of initiality: all nodes are considered to be non-initial.	Distinction between initial and non-initial move.


Clearly sequences of the form $\text{ext}(t)$ verify the requirements (w1) to (w5): For (w1), the initial node becomes \diamond . Explicit justification (w4) holds since we have added pointers to $@/\Sigma$ -nodes. Finally, alternation (w3), well-bracketing (w4) and visibility (w5) of the traversal t (Prop. 4.1.1) are preserved the extension operation (where visibility is defined with respect to the appropriate notion of P-view and O-view).

Clearly, $n \in t \upharpoonright^+ r_0 \iff \neg(n \notin t \upharpoonright^+ r_0)$ thus property (w6) holds. So does the switching condition (w7): Indeed if $t = \dots m \cdot n$ where $n \in N_{\text{var}} \cup L_{\lambda} \cup N_{@} \cup N_{\Sigma}$ and $m \in N_{\lambda} \cup L_{\text{var}} \cup L_{\Sigma} \cup L_{@}$ then, by definition of $t \upharpoonright^+ r_0$, m appears in $t \upharpoonright^+ r_0$ if and only if n does. For (w8): using the analogy of the preceding table, since all nodes are considered “non-initial” in $\text{ext}(t)$, this condition can be stated as:

(w8) Suppose m justifies n in $\text{ext}(t)$. Then $n \in t \upharpoonright^+ r_0$ if and only if $m \in t \upharpoonright^+ r_0$.

Unfortunately, as we have seen previously, the direct implication does not hold in general! (Indeed, a variable node can very well appear in $t \upharpoonright^+ r_0$ even though its justifier does not.) Consequently, the proof of Proposition 4.1.3 cannot be directly reused in our setting. A weaker version of condition (w8) holds however: we know by Lemma 4.1.11(i) that if r_0 occurs before n ’s justifier then n appears in $t \upharpoonright^+ r_0$ if and only if its justifier does. This condition turns out to be sufficient to reuse most of the proof of Proposition 4.1.3 [HO00].

We reproduce here some definition used in this proof. Let s be a position of the game $A \rightarrow B$.

A bounded segment is a segment θ of s of the form $\overset{x}{\circ} \dots \overset{y}{\bullet}$. If x is in A , and hence also y , then θ is an A -bounded segment. Respectively if x and y  in B then it is a B -bounded segment. By an abuse of notation we define $\lceil \theta \rceil B^\top$ to be the subsequence of $\lceil s \rceil_{s \leq y} \upharpoonright B^\top$ consisting only of moves in θ appearing after (and not including) x .

We then have:

Lemma 4.1.15. [HO00, Lemma A.3] *Let θ be an A -bounded segment in s with end-moves x and y .*

(i) $\lceil \theta \rceil B^\top = \overset{p_r}{\circ} \overset{q_r}{\bullet} \dots \overset{p_1}{\circ} \overset{q_1}{\bullet}$ for some $r \geq 0$. Note that each segment $p_i \dots q_i$ is B -bounded in s , for $1 \leq i \leq r$.

(ii) For any P-move m in θ which appear in $\lfloor s \rfloor_{s < y}$, m does not belong to any of the B -bounded segment $p_i \dots q_i$ for $1 \leq i \leq r$.

This Lemma assumes that the segment θ verifies the assumptions (w1) to (w8). As we have seen, (w8) does not always hold for extended traversals. But using our analogy with extended traversals, a segment θ is “A-bounded” if θ is bounded by two nodes appearing in $t \upharpoonright^+ r_0$. This can only happen if r_0 occurs before θ in t or if θ ’s left bound is r_0 . Thus the condition (w8) necessarily holds (at least for the nodes of the segment θ). The previous lemma thus translates into:

Lemma 4.1.16. *Let t be a traversal and θ be a segment of $\text{ext}(t)$ bounded by nodes x and y appearing in $t \upharpoonright^+ r_0$.*

- (i) $\ulcorner \theta \upharpoonright^+ r_0 \urcorner^e = p_r \cdot \overleftarrow{q_r} \dots p_1 \cdot \overleftarrow{q_1}$ for some $r \geq 0$ where $p_i \in N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\otimes$ and $q_i \in N_{\text{var}} \cup L_\lambda \cup N_\otimes \cup N_\Sigma$, for $1 \leq i \leq r$.
- (ii) For any node m in $N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\otimes$ occurring in θ and appearing in $\ulcorner \text{ext}(t) \urcorner_{<y \downarrow e}$, m does not belong to any of the segment $p_i \dots q_i$ for $1 \leq i \leq r$.

We can now show the analogy of Proposition 4.1.3 in the context of extended traversals:

Proposition 4.1.4. *Let t be a traversal and r_0 be an occurrence of some lambda node r' . If $\text{ext}(t)$ ’s last node appears in $t \upharpoonright^+ r_0$ then $\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 \sqsubseteq \ulcorner \text{ext}(t \upharpoonright^+ r_0) \urcorner^e$.*

Proof. By Eq. 4.3 we can equivalently show that: $\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 \sqsubseteq \ulcorner \text{ext}(t) \upharpoonright^+ r_0 \urcorner^e$. By induction on the length of t . The base case is immediate. For the inductive case, we do a case analysis:

- $t = t' \cdot r_0$. We have $\text{ext}(t) \upharpoonright^+ r_0 = r_0$ and $\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 = r_0 = \ulcorner \text{ext}(t) \upharpoonright^+ r_0 \urcorner^e$.
- $t = t' \cdot n$ with $n \in N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\otimes$ where n is not the occurrence r_0 .

There are two cases.

- Suppose that the last node in t' appears in $t \upharpoonright^+ r_0$. Then by the I.H. we have $\ulcorner \text{ext}(t') \urcorner^e \upharpoonright^+ r_0 \sqsubseteq \ulcorner \text{ext}(t') \upharpoonright^+ r_0 \urcorner^e$.

$$\begin{aligned}
 \ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 &= \ulcorner \text{ext}(t') \urcorner^e \upharpoonright^+ r_0 \cdot n && \text{(P-view for extended justified sequences of nodes of } M) \\
 &\sqsubseteq \ulcorner \text{ext}(t') \upharpoonright^+ r_0 \urcorner^e \cdot n && \text{(induction hypothesis)} \\
 &= \ulcorner \text{ext}(t') \upharpoonright^+ r_0 \cdot n \urcorner^e && \text{(P-view for extended justified sequences of nodes of } M^{(r')}: n \text{ belongs to } \tau(M^{(r')}) \text{ by Lemma 4.1.12)} \\
 &= \ulcorner \text{ext}(t' \cdot n) \upharpoonright^+ r_0 \urcorner^e && (n \text{ occurs in } t \upharpoonright^+ r_0) \\
 &= \ulcorner \text{ext}(t) \upharpoonright^+ r_0 \urcorner^e && \text{(definition of } t).
 \end{aligned}$$

- Suppose that the last node y_1 in t' does not appear in $t \upharpoonright^+ r_0$. Let \underline{m} be the last node preceding m in $\ulcorner \text{ext}(t) \urcorner^e$ that appears in $t \upharpoonright^+ r_0$. Then for some $q \geq 0$ we have

$$\ulcorner \text{ext}(t) \urcorner^e = \ulcorner \text{ext}(t) \urcorner_{\leq \underline{m}}^e \cdot \underbrace{x_q \cdot \overleftarrow{y_q} \dots x_1 \cdot \overleftarrow{y_1}}_{\text{all appear in } t \upharpoonright^+ r_0 \cdot m}$$

where the x_i s are in $N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\otimes$ and the y_i s are in $N_{\text{var}} \cup N_\Sigma \cup N_\otimes \cup L_\lambda$.

Therefore the sequence $\text{ext}(t)$ must be of the following form:

$$\text{ext}(t)_{\leq \underline{m}} \cdot \underbrace{x_q \dots y_q}_{\theta_q} \dots \underbrace{x_1 \dots y_1}_{\theta_1} \cdot m$$

where each segment θ_i is bounded by nodes appearing in $t \upharpoonright^+ r_0$. By Lemma 4.1.16, when computing the P-view of $\text{ext}(t)$, pointers going from a segment θ to a node outside the segment are never followed! In other words:

$$\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 = \ulcorner \text{ext}(t) \urcorner_{\leq \underline{m}}^e \upharpoonright^+ r_0 \urcorner^e \cdot \ulcorner \theta_q \upharpoonright^+ r_0 \urcorner^e \cdot \dots \cdot \ulcorner \theta_1 \upharpoonright^+ r_0 \urcorner^e \cdot m .$$

Hence:

$$\begin{aligned}
\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 &= \ulcorner \text{ext}(t) \urcorner_{\leq m}^e \upharpoonright^+ r_0 \cdot n \\
&\sqsubseteq \ulcorner \text{ext}(t) \urcorner_{\leq m} \upharpoonright^+ r_0 \urcorner^e \cdot n && \text{(I.H.)} \\
&\sqsubseteq \ulcorner \text{ext}(t) \urcorner_{\leq m} \upharpoonright^+ r_0 \urcorner^e \cdot \ulcorner \theta_q \upharpoonright^+ r_0 \urcorner^e \dots \ulcorner \theta_1 \upharpoonright^+ r_0 \urcorner^e \cdot n \\
&= \ulcorner \text{ext}(t) \urcorner \upharpoonright^+ r_0 \urcorner^e && \text{(by the previous equation).}
\end{aligned}$$

- $t = t' \cdot \overline{m \cdot u \cdot n}$ where $n \in N_{\text{var}} \cup N_{\Sigma} \cup N_{\text{@}} \cup L_{\lambda}$. We have $m \in N_{\lambda} \cup L_{\text{var}} \cup L_{\Sigma} \cup L_{\text{@}}$.

Suppose that r_0 appears in $t' \cdot m$, then since n appears in $t \upharpoonright^+ r_0$, by Lemma 4.1.11(i) so does m . Thus we can apply the I.H. on $t' \cdot m$:

$$\begin{aligned}
\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 &= \ulcorner \text{ext}(t') \cdot \overline{m \cdot u \cdot n} \urcorner_M^e \upharpoonright^+ r_0 && \text{(definition of } t) \\
&= (\ulcorner \text{ext}(t') \cdot \overline{m \cdot u \cdot n} \urcorner^e \upharpoonright^+ r_0) \upharpoonright^+ r_0 && \text{(P-eview computation in } M) \\
&= \ulcorner \text{ext}(t' \cdot m) \urcorner^e \upharpoonright^+ r_0 \cdot n && (n \text{ appears in } t \upharpoonright^+ r_0) \\
&\sqsubseteq \ulcorner (\text{ext}(t' \cdot m)) \urcorner \upharpoonright^+ r_0 \urcorner^e \cdot n && \text{(induction hypothesis on } t' \cdot m) \\
&= \ulcorner \text{ext}(t') \urcorner \upharpoonright^+ r_0 \cdot \overline{m \cdot u \cdot n} \urcorner^e && (m \text{ appears in } t \upharpoonright^+ r_0) \\
&= \ulcorner \text{ext}(t') \urcorner \upharpoonright^+ r_0 \cdot \overline{m \cdot (\text{ext}(u) \upharpoonright^+ r_0) \cdot n} \urcorner^e && \text{(P-eview in } M^{(r')}, \text{ nodes in } \\
&&& m \cdot (\text{ext}(u) \upharpoonright^+ r_0) \cdot n \text{ are all in } V^{(r')}) \\
&= \ulcorner (\text{ext}(t') \cdot \overline{m \cdot \text{ext}(u) \cdot n}) \urcorner \upharpoonright^+ r_0 \urcorner^e && (m \text{ and } n \text{ both appear in } t \upharpoonright^+ r_0) \\
&= \ulcorner \text{ext}(t) \urcorner \upharpoonright^+ r_0 \urcorner^e && \text{(definition of } t).
\end{aligned}$$

Suppose that r_0 appears in u then:

$$\begin{aligned}
\ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 &= \ulcorner \text{ext}(t' \cdot m) \urcorner^e \upharpoonright^+ r_0 \cdot n \\
&= n && (r_0 \text{ occurs after } t' \cdot m) \\
&\sqsubseteq \ulcorner (\text{ext}(t' \cdot m)) \urcorner \upharpoonright^+ r_0 \urcorner^e \cdot n \\
&= \ulcorner \text{ext}(t) \urcorner \upharpoonright^+ r_0 \urcorner^e .
\end{aligned}$$

□

We are now in a position to prove Proposition 4.1.2:

Proof of Proposition 4.1.2. We have:

$$\begin{aligned}
\ulcorner t \urcorner \upharpoonright^+ r_0 &= \ulcorner \text{ext}(t) \urcorner \upharpoonright^+ r_0 && \text{by Eq. 4.4} \\
&\sqsubseteq \ulcorner \text{ext}(t) \urcorner^e \upharpoonright^+ r_0 && \text{by Eq. 4.6} \\
&\sqsubseteq \ulcorner \text{ext}(t \upharpoonright^+ r_0) \urcorner^e && \text{by Proposition 4.1.4} \\
&= w \cdot \ulcorner \text{ext}(t \upharpoonright^+ r_0) \urcorner && \text{for some } w, \text{ by Eq. 4.6} \\
&= w \cdot \ulcorner t \urcorner \upharpoonright^+ r_0 \urcorner && \text{by Eq. 4.4.}
\end{aligned}$$

Thus $\ulcorner t \urcorner \upharpoonright^+ r_0 \sqsubseteq w \cdot \ulcorner t \urcorner \upharpoonright^+ r_0 \urcorner$. By definition of the operator $\ulcorner \cdot \urcorner$, both $\ulcorner t \urcorner \upharpoonright^+ r_0$ and $\ulcorner t \urcorner \upharpoonright^+ r_0 \urcorner$ start with the occurrence r_0 , thus this implies $\ulcorner t \urcorner \upharpoonright^+ r_0 \sqsubseteq \ulcorner t \urcorner \upharpoonright^+ r_0 \urcorner$. □

Example 4.1.10. Take $\varphi : ((o, o), o), e : o \vdash \varphi(\lambda x. (\lambda \psi. \varphi(\lambda x'. (\lambda y. \psi(\lambda z. z)))(\varphi(\lambda x''. x'))))(\lambda u. ue))$. The computation tree is represented below together with an example of traversal t :

Since n is a variable node appearing in $t \upharpoonright^+ r_0$, by definition of $t \upharpoonright^+ r_0$ its immediate predecessor $\lambda\bar{\xi}$ must occur in $t \upharpoonright^+ r_0$ and therefore must be the last occurrence in $t' \upharpoonright^+ r_0$. Thus we can use the rule (Lam) in $\tau(M^{(r')})$ to produce the traversal $u = (t' \upharpoonright^+ r_0) \cdot n$ of $M^{(r')}$.

We have $t \upharpoonright^+ r_0 = (t' \upharpoonright^+ r_0) \cdot n$, but in order to state that $u = t \upharpoonright^+ r_0$ it remains to prove that n has the same link in $t \upharpoonright^+ r_0$ and in u .

Suppose $n \in N_{\text{@}} \cup N_{\Sigma}$ then n has no justifier in both u and $t \upharpoonright^+ r_0$. Otherwise $n \in N_{\text{var}}$. Let m_u denote the occurrence in t of n 's justifier in u , m_t for the occurrence in t of n 's justifier in t , and m for the occurrence in t of n 's justifier in $t \upharpoonright^+ r_0$. We want to show that $m_u = m$. By the rule (Var), m_u is defined as the only occurrence of n 's enabler in $\ulcorner t' \upharpoonright^+ r_0 \urcorner$ and m_t is the only occurrence of n 's enabler in $\ulcorner t' \urcorner$.

If r_0 occurs before m_t then by Lemma 4.1.11(ii), m_t appears in $t \upharpoonright^+ r_0$ thus by definition of $_ \upharpoonright^+$ we have $m = m_t$. Moreover, since m_t appears in $t \upharpoonright^+ r_0$, it must appear after r_0 by Lemma 4.1.11(i.a), thus since it is in the P-view at t' , it must be in $\ulcorner t' \urcorner_{\geq r_0}$ which is equal to $\ulcorner t' \upharpoonright^+ r_0 \urcorner$ by Lemma 4.1.11(i.b) Hence we necessarily have $m_u = m_t$ (since r' occurs only once in the P-view $\ulcorner t' \upharpoonright^+ r_0 \urcorner$).

If r_0 occurs after m_t then m_t does not appear in $t \upharpoonright^+ r_0$ thus $m = r_0$ by definition of $_ \upharpoonright^+$. Moreover by Lemma 4.1.11(i), n 's binder occurs in the path from r' to the root \otimes . Thus n is a free variable in $\tau(M^{(r')})$ and consequently the only enabler of n occurring in $\ulcorner t' \upharpoonright^+ r_0 \urcorner$ is necessarily r_0 : $m_u = r_0$. This proves the equality $t \upharpoonright^+ r_0 = u$ and thus $t \upharpoonright^+ r_0$ is a valid traversal of $M^{(r')}$.

- (App) $t = \dots \lambda\bar{\xi} \cdot @ \cdot n$. Since n appears in $t \upharpoonright^+ r_0$, so does $@$ (by definition of $t \upharpoonright^+ r_0$). Hence $@$ is the last occurrence in $t' \upharpoonright^+ r_0$. By the induction hypothesis, $t' \upharpoonright^+ r_0$ is a traversal of $\tau(M^{(r')})$ therefore we can use the rule (App) in $\tau(M^{(r')})$ to produce the traversal $(t' \upharpoonright^+ r_0) \cdot n = t \upharpoonright^+ r_0$ of $M^{(r')}$.

- (Value $^{\text{@} \mapsto \lambda}$) Take $t = t' \cdot \lambda\bar{\xi} \cdot @ \dots v_{\text{@}} \cdot v_{\lambda\bar{\xi}}$.

The occurrence $v_{\lambda\bar{\xi}}$ appears in $t \upharpoonright^+ r_0$ therefore since r_0 is not a lambda node, its justifier $\lambda\bar{\xi}$ also appear in $t \upharpoonright^+ r_0$. Moreover since $@$ and $v_{\text{@}}$ are her. by $\lambda\bar{\xi}$, they must also appear in $t \upharpoonright^+ r_0$. By the induction hypothesis $t' \upharpoonright^+ r_0$ is a traversal of $\tau(M^{(r')})$ therefore since the occurrence $\lambda\bar{\xi}$, $@$, $v_{\text{@}}$, $v_{\lambda\bar{\xi}}$ all appear in $t \upharpoonright^+ r_0$ we can use the rule (Value $^{\text{@} \mapsto \lambda}$) in $M^{(r')}$ to form the traversal $(t' \upharpoonright^+ r_0) \cdot n = t \upharpoonright^+ r_0$ of $M^{(r')}$.

- (Value $^{\lambda \mapsto \text{@}}$) Take $t = t' \cdot @ \cdot \lambda\bar{\xi} \dots v_{\lambda\bar{\xi}} \cdot v_{\text{@}}$. Again, since $v_{\text{@}}$ appears in $t \upharpoonright^+ r_0$, necessarily the occurrences $@$, $\lambda\bar{\xi}$, $v_{\lambda\bar{\xi}}$ and $v_{\text{@}}$ must all appear in $t \upharpoonright^+ r_0$. Hence using the induction hypothesis and the rule (Value $^{\lambda \mapsto \text{@}}$) in $M^{(r')}$ we obtain that $t \upharpoonright^+ r_0$ is a traversal of $M^{(r')}$.

- (Value $^{\text{var} \mapsto \lambda}$) Take $t = t' \cdot \lambda\bar{\xi} \cdot x \dots v_x \cdot v_{\lambda\bar{\xi}}$. Since $v_{\lambda\bar{\xi}}$ is in $t \upharpoonright^+ r_0$, so must be x , v_x and $\lambda\bar{\xi}$, by definition of $t \upharpoonright^+ r_0$. Hence we can use the I.H. to form the traversal $t \upharpoonright^+ r_0$ of $M^{(r')}$.

- (InputValue) Take $t = t_1 \cdot x \cdot t_2 \cdot v_x$ for some $v \in \mathcal{D}$ where x is the pending node in $t_1 \cdot x \cdot t_2$ and $x \in N_{\text{var}}^{\text{@} \vdash}$.

Since v_x appears in $t \upharpoonright^+ r_0$, so does x hence by Lemma 4.1.9, x is also the pending node in $(t_1 \cdot x \cdot t_2) \upharpoonright^+ r_0$. Furthermore since $M^{(r')}$ is a subterm of M , x is necessarily an input-variable node in $\tau(M^{(r')})$. Hence we can conclude using the I.H. and the rule (InputValue).

- (InputVar) Take $t = t' \cdot n$ where $n \in N_{\lambda}$ points to an occurrence of its parent node $y \in N_{\text{var}}^{\text{@} \vdash}$ in $\ulcorner t \urcorner$. By Lemma 4.1.8(a), y must also appear in $t \upharpoonright^+ r_0$, therefore y also occurs in $\ulcorner t \upharpoonright^+ r_0 \urcorner \sqsubseteq \ulcorner t \urcorner \upharpoonright^+ r_0$. Hence we can conclude using the rule (InputVar) in $M^{(r')}$.

- (Var) Take $t = t' \cdot p \cdot \lambda\bar{\xi} \dots x_i \cdot \lambda\bar{\eta}_i$ for some variable x_i in $N_{\text{var}}^{\text{@} \vdash}$. If $\lambda\bar{\eta}_i$ is the occurrence r_0 then the traversal $t \upharpoonright^+ r_0 = r_0$ can be formed using the rule (Root).

Suppose that $\lambda\bar{\eta}_i$ is not the occurrence r_0 . Then both $\lambda\bar{\eta}_i$ and its justifier p must appear in $t \upharpoonright^+ r_0$. The nodes $\lambda\bar{\xi}$ and x_i , however, do not necessarily appear in $t \upharpoonright^+ r_0$.

Consider the node $@$ that initiates the thread of $\lambda\bar{\eta}_i$.

- Suppose that r_0 precedes $@$ in t then by Lemma 4.1.14(i), the nodes $\lambda\bar{\eta}_i$, p , $\lambda\bar{\xi}$ and x_i as well as $@$ all appear in $t \upharpoonright^+ r_0$. Moreover since $@$ appear in $t \upharpoonright^+ r_0$, it must be an occurrence of

an application node that appear in the subtree rooted at r' thus $@ \in N_{\text{var}}^{r' \vdash}$. Hence we can use the rule (Var) in $M^{(r')}$ to form the traversal $t \vdash^+ r_0$ of $M^{(r')}$.

- Suppose that $@$ precedes r_0 in t then by Lemma 4.1.14(ii), p is necessarily an input variable node in $\tau(M^{(r')})$.

We have $p \in \perp t \vdash^+ r_0 \sqsubseteq \perp t \vdash^+ r_0 \perp$ by Proposition 4.1.2. Furthermore we can easily check that the last node in $\text{ip}(t \vdash^+ r_0)$ is necessarily in $N_{\text{var}} \cup L_\lambda$ (using alternation and the fact that if an occurrence in $N_\lambda \cup L_{\text{var}} \cup L_{@} \cup L_\Sigma \cup N_{@} \cup N_\Sigma$ appears in $t \vdash^+ r_0$ then so does its immediate successor). Hence we can make use of the rule (InputVar) in $M^{(r')}$ (in its alternative form) to produce the traversal $t \vdash^+ r_0$ of $M^{(r')}$.

- (Value $^{\lambda \mapsto \text{var}}$) Take $t = t' \cdot y \cdot \lambda \bar{\xi} \dots v_{\lambda \bar{\xi}} \cdot v_y$ for some variable y in $N_{\text{var}}^{@ \vdash}$.

The proof is similar to the previous case using the rule (InputValue) instead of (InputVar) in the second subcase.

- $(\Sigma)/(\Sigma\text{-var})$ The proof is similar to the case (App) and (Var).
- $(\Sigma\text{-Value})$ The proof is similar to the case (Value $^{\lambda \mapsto \text{var}}$). □

The following Lemma will become useful in the proof of the Correspondence Theorem:

Lemma 4.1.17. *Let t be a traversal and r_0 be an occurrence of a lambda node r' . We have*

$$(t \vdash^+ r_0)^* = t^* \upharpoonright V^{(r')} \upharpoonright r_0 .$$

Proof. By the previous Lemma, $t \vdash^+ r_0$ is indeed a traversal (of $\tau(M^{(r')})$) thus the expression “ $(t \vdash^+ r_0)^*$ ” is well-defined. We show the result by induction on t : It is true for the empty traversal. Take $t = t' \cdot n$.

If n belongs to $V_{@} \cup V_\Sigma$ then

$$((t' \cdot n) \vdash^+ n_0)^* = (t' \vdash^+ n_0)^* \cdot \begin{cases} n, & \text{if } n \text{ appears in } t \vdash^+ n_0; \\ \epsilon, & \text{otherwise.} \end{cases}$$

$$\text{and } ((t' \cdot n)^* \upharpoonright V^{(r')}) \upharpoonright n_0 = (t'^* \upharpoonright V^{(r')}) \upharpoonright n_0 \cdot \begin{cases} n, & \text{if } n \text{ is her. just. by } n_0 \text{ in } t^* \upharpoonright V^{(r')}; \\ \epsilon, & \text{otherwise.} \end{cases}$$

Since $t^\omega \notin V_{@} \cup V_\Sigma$, by Lemma 4.1.13 we have that n is her. just. by n_0 in $t^* \upharpoonright V^{(r')}$ if and only if n appears in $t \vdash^+ n_0$. Hence we can conclude using the I.H. on t' .

If n does not belong to $V_{@} \cup V_\Sigma$ then

$$\begin{aligned} ((t' \cdot n) \vdash^+ n_0)^* &= (t' \vdash^+ n_0)^* \\ &= (t'^* \upharpoonright V^{(r')}) \upharpoonright n_0 && \text{(by the I.H. on } t') \\ &= ((t' \cdot n)^* \upharpoonright V^{(r')}) \upharpoonright n_0 && \square \end{aligned}$$

Consequently, by Lemma 4.1.7, if $t^\omega \notin V_{@} \cup V_\Sigma$ then $t \vdash^+ r_0 = (t^* \upharpoonright r_0) + \Sigma + @$.

4.1.3.9 O-view and P-view projection with respect to root

Lemma 4.1.18 (O-view projection with respect to the root). *Let t be a non-empty traversal of M and r denote the only occurrence of $\tau(M)$'s root in t . If t^ω appears in $t \upharpoonright r$ then:*

$$\perp t \upharpoonright r \perp = \perp t \perp \upharpoonright r = \perp t \perp .$$

Proof. It follows immediately from the fact that, by Lemma 4.1.6, all the occurrences in $\perp t \perp$ belong to the same thread and therefore are all hereditarily justified by r . □

Lemma 4.1.19 (P-view projection with respect to the root). *Let t be a non-empty traversal of M and r denote the only occurrence of $\tau(M)$'s root in t . If t^ω appears in $t \upharpoonright r$ then:*

$$\ulcorner t \urcorner \upharpoonright r \sqsubseteq \ulcorner t \urcorner \upharpoonright r^\neg .$$

Proof. We just sketch the proof. We proceed exactly in the same way as for the proof of Proposition 4.1.2. Again we establish an analogy between traversals and plays of game semantics:

Traversal setting	Game-semantic setting
Traversal t	Play s
Nodes in $n \in N_\lambda \cup L_{\text{var}} \cup L_\Sigma \cup L_\oplus$	O-moves \bullet
Nodes in $n \in N_{\text{var}} \cup L_\lambda \cup N_\oplus \cup N_\Sigma \cup \{\diamond\}$	P-moves \circ
P-view $\lceil t \rceil$	P-view $\lceil s \rceil$
O-view $\lfloor t \rfloor$	O-view $\lfloor s \rfloor$
Occurrence n her. just. by r in t	Occurrence $n \in B$
Occurrence n not her. just. by r in t	Occurrence $n \in A$
No notion of initiality: all nodes are considered to be non-initial.	Distinction between initial and non-initial move.

Clearly the conditions (w1) to (w8) hold. Hence we can reuse Proposition 4.3 from [HO00] which gives the desired result. \square

The downside of the previous result is that it does not give us an equality. In the particular case where interpreted constants are well behaved, however, if we consider the subsequence of a traversal consisting of unanswered nodes only, then we obtain a nice equality:

Lemma 4.1.20. *Suppose that M is in β -normal form and all the Σ -constants are well-behaved. Let t be a non-empty traversal of M and r denote the only occurrence in t of $\tau(M)$'s root.*

(a) *If t 's last occurrence is not a leaf then $\lceil t \rceil \upharpoonright r = \lceil ?(t) \rceil \upharpoonright r = \lceil ?(t \upharpoonright r) \rceil = \lceil ?(t \upharpoonright r) \rceil$;*

(b) *If t 's last occurrence is not a leaf and is hereditarily justified by r then $\lceil t \rceil \upharpoonright r = \lceil t \upharpoonright r \rceil$.*

Proof. (a) It is easy to show that $?(t) \upharpoonright r = ?(t \upharpoonright r)$. This implies the second equality. The third equality can be shown by an easy induction and by observing that in a traversal reduction, variable occurrences are always immediately preceded by a lambda node (and not by a leaf). We show the first equality by induction. The base case $t = \epsilon$ is trivial. Consider a traversal t and suppose that the property is verified for all traversals shorter than t . Observe that since t contains at most a single occurrence r of the root \oplus , an occurrence n in t is hereditarily justified by r if and only if the corresponding node in $\tau(M)$ is hereditarily enabled by \oplus . Thus $t \upharpoonright r = t \upharpoonright N^{\oplus \vdash}$. We do a case analysis on t 's last node:

- $t^\omega \in N_\oplus$. This case does not happen since M is β -normal.
- $t = t' \cdot n$ with $n \in N_{\text{var}} \cup N_\Sigma$ then t'^ω is not a leaf (otherwise n would also be a leaf by rule (Value)) thus we can use the I.H. on t' which, by an easy calculation, gives the desired equality.

Suppose that t^ω is a lambda node. There are three subcases:

- $t^\omega \in N_\lambda^{\oplus \vdash}$. Since the term is in β -normal form, there is no \oplus -node in $\tau(M)$ so the rules (App) and (Var) are unused, hence this case does not happen.
- $t^\omega \in N_\lambda^{N_\Sigma \vdash}$. We have $t = t' \cdot \overbrace{m \cdot u \cdot n}^{\text{arc}}$ with $n \in N_\lambda^{N_\Sigma \vdash}$ and $m \in N_{\text{var}} \cup N_\Sigma$. The occurrence n is necessarily visited with a (Σ) -rule. Since, by assumption, these rules are well-behaved we have $?(u) = \epsilon$. Hence:

$$\begin{aligned}
 \lceil t \rceil \upharpoonright r &= \lceil t' \cdot \overbrace{m \cdot u \cdot n}^{\text{arc}} \rceil \upharpoonright r && \text{(Def. of } t) \\
 &= (\lceil t' \rceil \cdot \overbrace{m \cdot n}^{\text{arc}}) \upharpoonright r && \text{(P-view computation)} \\
 &= \lceil t' \rceil \upharpoonright r && (m, n \notin N^{\oplus \vdash}) \\
 &= \lceil ?(t') \rceil \upharpoonright r && \text{(induction hypothesis)} \\
 &= \lceil ?(t' \cdot \overbrace{m \cdot n}^{\text{arc}}) \rceil \upharpoonright r && (m, n \notin N^{\oplus \vdash}) \\
 &= \lceil ?(t' \cdot \overbrace{m \cdot u \cdot n}^{\text{arc}}) \rceil \upharpoonright r && (?(u) = \epsilon) \\
 &= \lceil ?(t) \rceil \upharpoonright r && \text{(def. of } t \text{ \& } u = \epsilon).
 \end{aligned}$$

- $t^\omega \in N_\lambda^{\otimes \perp}$. If $t = r$ then the result holds trivially. Otherwise $t = t' \cdot m \cdot u \cdot n$ for some $n \in N_\lambda^{\otimes \perp}$. An easy calculation using the induction hypothesis on $t' \cdot m$ shows the desired equality.

(b) If t 's last occurrence is hereditarily justified by r then the last occurrence of $t \upharpoonright r$ is precisely the last occurrence of t and is therefore not a leaf. In a traversal reduction, variable nodes are immediately preceded by lambda nodes thus since the last node in $t \upharpoonright r$ is not a leaf, an easy induction shows that all the nodes in $\ulcorner t \upharpoonright r \urcorner$ are not leaves. Consequently $\ulcorner t \upharpoonright r \urcorner = \ulcorner t \upharpoonright r \urcorner$. \square

The hypothesis that the term is beta-normal is crucial in this Lemma. Take for instance the term $\lambda x^o f^{(o,o)}(\lambda y^o.f y)x$. A possible traversal is

$$t = \lambda x f \cdot @ \cdot \lambda y \cdot f \cdot \lambda \cdot y \cdot \lambda \cdot x .$$

But $\ulcorner t \urcorner \upharpoonright r = \lambda x f \cdot x$ is only a strict subsequence of $\ulcorner t \upharpoonright r \urcorner = \lambda x f \cdot f \cdot \lambda \cdot x$.

4.2 Game semantics correspondence

We work in the general setting of an applied simply typed lambda-calculus with a given set of higher-order constants Σ . The operational semantics of these constants is given by certain reduction rules. We assume that a fully abstract model of the calculus is provided by means of a category of well-bracketed games. For instance, if Σ consists of the PCF constants then we work in the category of games and innocent well-bracketed strategies [HO00, AMJ94]. A strategy is commonly defined in the literature as a set of plays closed by even-length prefixing. For our purpose, however, it is more convenient to represent strategies using *prefix-closed* set of plays. This will spare us some considerations on the parity of traversal length when showing the correspondence between traversals and game semantics. For the rest of the section we fix a simply typed term $\Gamma \vdash M : T$. We write $\llbracket \Gamma \vdash M : T \rrbracket$ for its strategy denotation (in the standard cartesian closed category of games and innocent strategies [AMJ94, HO00]). We use the notation $\text{Pref}(S)$ to denote the prefix-closure of the set S .

4.2.1 Interaction game semantics

In standard game semantics, terms are denoted by strategies that are computed inductively on the structure of the term: calculating the denotation of a term boils down to performing the composition of strategies denoting some of its subterms. Strategy composition consists of a CSP-like “composition + hiding” operation where all the internal moves are hidden.

It is possible to use an alternative notion of composition where the internal moves are not hidden. Game model based on such notion of composition have appeared in the literature under the name *revealed semantics* [Gre04] and *interaction semantics* [DGL05]. In such game models, the denotation is computed inductively on the syntax of the term as in the standard game semantics, but certain internal moves may be uncovered after composition. There is not just one interaction semantics as one may desire to hide/uncover different internal moves. Moreover, since the denotation is computed inductively on the syntax of the term, the denotation is necessarily sensitive to the syntax and therefore such model cannot provide a full abstraction of the language considered. However this semantics will prove to be useful to identify a correspondence between the game semantics of a term and the traversals of its computation tree.

This section presents a general setting in which interaction semantics can be defined. At the end of the section we will provide an example of such interaction semantics that is calculated inductively on the syntax of the η -long normal form of the term.

4.2.1.1 Revealed strategies

Definition 4.2.1.

- We call **interaction type tree** or just **interaction type**, a tree whose leaves are labelled with PCF simple types and inner nodes are labelled with symbol in $\{;, \langle -, - \rangle, \Lambda\}$.
Nodes labelled $;$ and $\langle -, - \rangle$ are binary nodes while nodes labelled by Λ are unary nodes. If T_1 and T_2 are interaction types we write $\langle T_1, T_2 \rangle$ to denote the interaction type obtained by attaching T_1 and T_2 to a $\langle -, - \rangle$ -node. Similarly we use the notations $T_1; T_2$ and $\Lambda(T_1)$.
- We define $\text{type}(T)$, the *underlying simple type* of T or just type of T as follows:
 - If T is a leaf then $\text{type}(T)$ is the type that labels it;
 - $\text{type}(\Lambda(T_1^{A \times B \rightarrow C})) = A \rightarrow (B \rightarrow C)$;
 - $\text{type}(\langle T_1^{C \rightarrow A}, T_2^{C \rightarrow B} \rangle) = C \rightarrow A \times B$;
 - $\text{type}(T_1^{A \rightarrow B}; T_2^{B \rightarrow C}) = A \rightarrow C$.

We sometimes indicate the type in exponent; for instance $T^{A \rightarrow B}$ if $\text{type}(T) = A \rightarrow B$. The type of a node of the tree T is then defined as the type of the subtree rooted at that node.

Note that it is straightforward to generalize the pairing operator to a p -tuple operator $\langle \Sigma_1, \dots, \Sigma_p \rangle$ for $p \geq 2$ (in which case the root of the interaction game has p children).

For the interaction type tree to be well-defined, it is required that types of children nodes are consistent with the kind of the parent node. For instance the two children nodes of a $;$ -node must be of type $A \rightarrow B$ and $B \rightarrow C$.

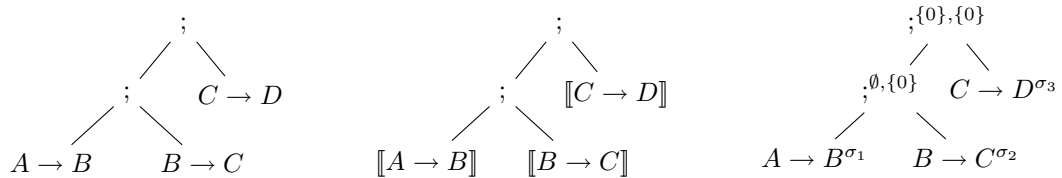
Let T be an interaction type tree. Each node of type A in T can be mapped to the (standard) game $\llbracket A \rrbracket$. By taking the image of T across this mapping we obtain a tree whose leaves and nodes are labelled by games. This tree, written $\langle\langle T \rangle\rangle$, is called an **interaction game**.

A **revealed strategy** Σ on the interaction game $\langle\langle T \rangle\rangle$ is a compositions of several standard strategies in which certain internal moves are not hidden. Formally:

Definition 4.2.2. A **revealed strategy** Σ on an interaction game $\langle\langle T \rangle\rangle$, written $\Sigma : \langle\langle T \rangle\rangle$, is an annotated interaction type tree T where

- each leaf $\llbracket A \rrbracket$ of T is annotated with a (standard) strategy σ on the game $\llbracket A \rrbracket$;
- each $;$ -node is annotated with two sets of indices $U_{\text{sup}}, U_{\text{prof}} \subseteq \mathbb{N}$ called respectively the *superficial* and *profound* uncovering indices.

Example 4.2.1. The diagrams below represent an interaction type tree T (left), the corresponding interaction game $\langle\langle T \rangle\rangle$ (middle) and a revealed strategy Σ (right):



A revealed strategy can also be written as an expression, for instance the strategy represented above is given by the expression $\Sigma = (\sigma_1;^{\emptyset, \{0\}} \sigma_2);^{\{0\}, \{0\}} \sigma_3$.

The intuition behind this definition is that if a $;$ -node has children $\Sigma_1 : \langle\langle A \rightarrow B \rangle\rangle$ and $\Sigma_2 : \langle\langle B \rightarrow C \rangle\rangle$ then the two sets of indices $U_{\text{sup}}, U_{\text{prof}}$ indicate which components of B should be uncovered when performing composition. The set U_{sup} permits one to uncover *superficial* internal moves (*i.e.*, those that are created by the top-level composition between Σ_1 and Σ_2), while the set U_{prof} can be used to cover/uncover the *profound* internal moves (*i.e.*, those that are already present in the revealed strategies Σ_1 and Σ_2). The precise definition of this uncovering will be unveiled in the definition of the *uncovered positions* of a revealed strategy in the next paragraph.

4.2.1.2 Uncovered play

The analogous of a play in the interaction semantics is called an *uncovered play*, it is a play containing internal moves. Each move may belong to several games from different nodes of the interaction game. The moves are implicitly tagged so that it is possible to retrieve in which component of the arena of which node-game the move belongs to.

Definition 4.2.3. The *set of possible moves* M_T of an interaction game $\langle\langle T \rangle\rangle$ is defined as \mathcal{M}_T / \sim_T , the quotient of the set \mathcal{M}_T by the equivalence relation $\sim_T \subseteq \mathcal{M}_T \times \mathcal{M}_T$ defined as follows: For a single leaf tree T labelled by a type A we define $\mathcal{M}_T = M_A$ and $\sim_T = id_{M_A}$; for other cases:

$$\begin{aligned}\mathcal{M}_{\Lambda(T)} &= \mathcal{M}_T + M_{type(\Lambda(T))} \\ \sim_{\Lambda(T)} &= (\sim_T \cup (type \Lambda(T) \leftrightarrow type T))^*\end{aligned}$$

$$\begin{aligned}\mathcal{M}_{\langle T_1^{C^1 \rightarrow A^1}, T_2^{C^2 \rightarrow B^2} \rangle} &= \mathcal{M}_{T_1} + \mathcal{M}_{T_2} + M_{C \rightarrow (A \times B)} \\ \sim_{\langle T_1^{C^1 \rightarrow A^1}, T_2^{C^2 \rightarrow B^2} \rangle} &= (\sim_{T_1} \cup \sim_{T_2} \cup (C^1 \leftrightarrow C) \cup (C^2 \leftrightarrow C) \cup (A^1 \leftrightarrow A) \cup (B^2 \leftrightarrow B))^*\end{aligned}$$

$$\begin{aligned}\mathcal{M}_{T_1^{A \rightarrow B}, T_2^{B \rightarrow C}} &= \mathcal{M}_{T_1} + \mathcal{M}_{T_2} + M_{A \rightarrow C} \\ \sim_{T_1^{A \rightarrow B}, T_2^{B \rightarrow C}} &= (\sim_{T_1} \cup \sim_{T_2} \cup (A^1 \leftrightarrow A) \cup (B^1 \leftrightarrow B^2) \cup (C \leftrightarrow C^2))^*\end{aligned}$$

where $A \leftrightarrow B$ denotes the canonical bijection between M_A and M_B for two isomorphic arenas A and B ; and R^* denotes the smallest superset of the relation R complete by transitivity, reflexivity and symmetry.

We call *internal move* of the game $\langle\langle T \rangle\rangle$, any \sim -class from M_T that does not contain any move from $M_{type(T)}$. We denote the set of all internal moves by M_T^{int} . The complement of M_T^{int} in M_T , called the set of *external moves*, is denoted by M_T^{ext} .

For any subgame A occurring in some node of the interaction game T , we write $M_{T,A}^{\text{int}}$ (resp. $M_{T,A}^{\text{ext}}$) for the subset of moves of M_T^{int} (resp. M_T^{ext}) consisting of \sim -classes containing some move in M_A .

A *justified interaction sequence* of moves on the interaction game $\langle\langle T \rangle\rangle$ is a sequence of moves from M_T together with pointers where each move in the sequence except the first one has a link attached to it pointing to some preceding move in the sequence.

Definition 4.2.4 (Projection). We define several projection operations over justified interaction sequences. Let s be a justified sequence of moves on the interaction game $\langle\langle T \rangle\rangle$.

- i. Let T' be a direct subtree of T . We define the projection $s \upharpoonright T'$ to be the subsequence of s consisting of moves \sim -equivalent to some move in $M_{T'}$. This operation may cause a move to “lose” its pointer. This situation corresponds to $(s \cdot m) \upharpoonright T' = (s \upharpoonright T') \cdot m$ where m ’s justifier does not appear in $s \upharpoonright T'$. In that case, a new link is reassigned to m that points to the only occurrence of an initial $\llbracket type(T') \rrbracket$ -move in $\lceil s \upharpoonright \llbracket type(T') \rrbracket \rceil$ (where $s \upharpoonright \llbracket type(T') \rrbracket$ is defined in iii.).

Note that since M_T is a set of equivalence classes with respect to \sim , the projection operator $_ \upharpoonright T'$ implicitly performs the “retagging” of the moves to the appropriate components of each game of the interaction game $\langle\langle T \rangle\rangle$.

- ii. Let T' be a non-direct subtree of T . We define the projection $s \upharpoonright T'$ as $(\dots (s \upharpoonright T^0) \upharpoonright \dots \upharpoonright T^{k-1}) \upharpoonright T^k$ where $T = T^0$ and $T' = T^k$ and for every $1 \leq l \leq k$, T^l is a direct subtree of T^{l-1} .
- iii. Let T' be some subtree of T and A be a sub-game of $\llbracket type(T') \rrbracket$. We define the projection operator $s \upharpoonright A$ to be the subsequence of $s \upharpoonright T'$ consisting of the \sim -classes that contain some move in M_A .

- iv. For any initial move m of the game $\llbracket \text{type}(T) \rrbracket$ occurring in s , $s \upharpoonright m$ is the subsequence of s consisting of moves that are *hereditarily justified* by that particular occurrence of m in $s \upharpoonright \text{type}(T)$.

By extension, we also define these operations on sets of justified interaction sequences.

Definition 4.2.5 (Legal uncovered positions). We recall that in the standard game semantics, the set of legal positions L_A of a game A is the set of justified sequences of moves from M_A respecting visibility and alternation. We define the set of **legal uncovered position** L_T of an interaction game $\langle\langle T \rangle\rangle$ as follows. If T is

- a leaf annotated by a type A then $L_T = L_A$;
- a unary node with child node T' then:

$$L_T = \{s \in \text{JustSeq}(T) \mid s \upharpoonright \text{type}(T) \in L_{\text{type}(T)} \wedge s \upharpoonright T' \in L_{T'}\};$$

- a binary node with children nodes T_1 and T_2 then:

$$L_T = \{s \in \text{JustSeq}(T) \mid s \upharpoonright \text{type}(T) \in L_{\text{type}(T)} \wedge s \upharpoonright T_1 \in L_{T_1} \wedge s \upharpoonright T_2 \in L_{T_2}\}.$$

where $\text{JustSeq}(T)$ denotes the set of justified interaction sequences on $\langle\langle T \rangle\rangle$.

We now define a revealed strategy by means of a set of uncovered positions the interaction game. The set of positions of an interaction strategy is computed from its constituent standard strategies using a bottom-up computation starting from the leaves of the tree representing the strategy. In particular, at each $;$ -node the composition operation of game semantics will be applied. But in contrast with the standard game semantics, however, internal moves will not be all hidden. But in accordance with standard game semantics (see definition of the projection $_ \upharpoonright A, C$ [AM98b]), when composing $\Sigma_l : T_l^{A \rightarrow B}$ with $\Sigma_r : T_r^{B \rightarrow C}$, the justification pointers are readjusted as follows: if an initial A-move a has link pointing to an initial B-move itself pointing to an initial C-move c then it is replaced by a link pointing directly to c . Thus if we take any interaction position u of $\Sigma_l; \Sigma_r$, by taking the subsequence of u consisting of nodes in A and C only (ignoring the moves in B as well as the internal moves coming from compositions that took place at deeper level in the interaction semantics) then we obtain a valid *standard* position of the standard strategy that underlies the interaction strategy $\Sigma_l; \Sigma_r$.

Additionally, given a position of an interaction strategy Σ and a sub-interaction strategy $\Sigma' : T'$ of $\Sigma : T$, we would like to be able to obtain a valid position of Σ' . At first this may seem problematic since during composition we have lost all the links going from initial A-moves to initial B-moves! Fortunately, these can be retrieved easily. Indeed, take an interaction sequence $u \cdot a \in \text{Int}(A, B, C)$ where a is an initial A-move, then by P-visibility in the game $\boxplus \rightarrow B$, a points to an initial B-move in $\ulcorner u \upharpoonright A, B \urcorner$. But the P-view, by definition, contains at most one initial B-move! Hence the original pointer associated to a can just be retrieved from the P-view $\ulcorner u \upharpoonright A, B \urcorner$. We now observe that this pointer-recovery procedure corresponds precisely to the way the projection operation was defined in Def. 4.2.4. Consequently, for any position u of $\Sigma_l; \Sigma_r$ where $\Sigma_l : T_l$ and $\Sigma_r : T_r$, the sequence $u \upharpoonright T_l$ is a valid position of Σ_l and $u \upharpoonright T_r$ is a valid position of Σ_r . More generally, this implies that for any subtree $\Sigma' : T'$ of $\Sigma : T$, and any position u of the interaction strategy Σ , $u \upharpoonright T'$ is a valid position of Σ' .

Definition 4.2.6. The set of **uncovered positions** of a revealed strategy is defined inductively on the structure of the annotated interaction type tree underlying the interaction strategy:

- *Leaf* labelled with type A and annotated by the strategy σ : The set of positions of the revealed strategy is precisely the set of positions of the standard strategy σ .
- *Currying*: $\Lambda(\Sigma' : \langle\langle T' \rangle\rangle) : \langle\langle T \rangle\rangle = \{u \in L_T \mid u \upharpoonright T' \in \Sigma'\}$.

- *Pairing*:

$$\langle \Sigma_1 : \langle \langle T_1 \rangle \rangle, \Sigma_2 : \langle \langle T_2 \rangle \rangle \rangle : \langle \langle T \rangle \rangle = \{u \in L_T \mid (u \upharpoonright T_1 \in \Sigma_1 \wedge u \upharpoonright T_2 = \epsilon) \vee (u \upharpoonright T_1 = \epsilon \wedge u \upharpoonright T_2 \in \Sigma_2)\}$$

where $\text{type}(T_1) = C \rightarrow A$, $\text{type}(T_2) = C \rightarrow B$ and $\text{type}(T) = C \rightarrow A \times B$.

- *Uncovered composition*: Take $\Sigma_1 : \langle \langle T_1 \rangle \rangle$ and $\Sigma_2 : \langle \langle T_2 \rangle \rangle$ then $\Sigma_1 \parallel \Sigma_2$ is defined on the game $\langle \langle T \rangle \rangle$ where $\text{type}(T) = A \rightarrow C$, $\text{type}(T_1) = A \rightarrow B_0 \times \dots \times B_l$ and $\text{type}(T_2) = B_0 \times \dots \times B_l \rightarrow C$ as

$$\begin{aligned} \Sigma_1 \parallel \Sigma_2 = \{u \in L_T \mid & u \upharpoonright T_2 \in \Sigma_2 \\ & \wedge \text{ for all occurrence } m \text{ in } u \text{ of an initial } \llbracket \text{type}(T_1) \rrbracket\text{-move,} \\ & \quad u \upharpoonright T_1 \upharpoonright m \in \Sigma_1 \\ & \wedge \text{ for any initial } m \text{ in } A, \text{ if } m \text{ is justified in } u \upharpoonright T_1 \text{ by} \\ & \quad b \in B_j, \text{ itself justified by } c \in C^2 \text{ in } u \upharpoonright T_2 \text{ then } m \text{ is} \\ & \quad \text{justified by } c \text{ in } u. \} \end{aligned}$$

- *Partially covered composition*:

$$\Sigma_1 ;^{U_{\text{sup}}, U_{\text{prof}}} \Sigma_2 = \{\text{cover}(u, \{0..l\} \setminus U_{\text{sup}}, \{0..l\} \setminus U_{\text{prof}}) \mid u \in \Sigma_1 \parallel \Sigma_2\}$$

where

$$\text{cover}(u, C_{\text{sup}}, C_{\text{prof}}) = u \upharpoonright M_T \setminus \left(\bigcup_{j \in C_{\text{sup}}} \overbrace{M_{T_1, B_j}^{\text{ext}} \cup M_{T_2, B_j}^{\text{ext}}}^{\text{superficial } B_j\text{-moves}} \cup \bigcup_{j \in C_{\text{prof}}} \overbrace{M_{T_1, B_j}^{\text{int}} \cup M_{T_2, B_j}^{\text{int}}}^{\text{profound } B_j\text{-moves}} \right).$$

Observe that $\Sigma_1 \parallel \Sigma_2 = \Sigma_1 ;^{\{0..l\}, \{0..l\}} \Sigma_2$.

In words, the *uncovered composition* $\Sigma_1 \parallel \Sigma_2$ is the set of uncovered plays obtained by performing the usual composition of the standard strategies underlying Σ_1 and Σ_2 while preserving the internal moves already in Σ_1 and Σ_2 as well as the internal move produced by the composition.

Whereas if B is the product game $B_0 \times \dots \times B_l$ then the *partially covered composition* $\Sigma_1 ;^{U_{\text{sup}}, U_{\text{prof}}} \Sigma_2$ only keeps the superficial internal moves from the component B_k for some $k \in U_{\text{sup}}$ and the profound internal moves from the component B_k for some $k \in U_{\text{prof}}$.

Lemma 4.2.1 (Complete interaction sequence). *Let u be an interaction sequence of some interaction strategy $\Sigma : \langle \langle T \rangle \rangle$ and suppose that the standard strategy denoting the leaves of Σ are all well-bracketed.*

Then for any node game A of T and interaction sequence $u \in \Sigma$ we have:

- i. $u \upharpoonright A$ is well-bracketed;
- ii. If $u \upharpoonright \text{type}(T)$ is complete (all question moves answered) then $u \upharpoonright A$ is complete.

Proof. By induction on the structure of the interaction game $\langle \langle T \rangle \rangle$. The base case is trivial. We only treat composition, the other cases being trivial: Let $u \in \Sigma_1 ;^U \Sigma_2$ for some $U \subseteq \mathbb{N}$ with $\Sigma_1 : \langle \langle T_1^{A \rightarrow B} \rangle \rangle$ and $\Sigma_2 : \langle \langle T_2^{B \rightarrow C} \rangle \rangle$.

i. During composition, pointers attached to answer moves are preserved with respect to \sim thus non-well-bracketing of $u \upharpoonright A \rightarrow C$ implies either non-well-bracketing of $u \upharpoonright A \rightarrow B$ or $u \upharpoonright B \rightarrow C$.

For ii., suppose $u \upharpoonright \text{type}(T) = q \overline{u} a$. By well-bracketing (i.) and since q and a belong to C we must have $u \upharpoonright B \rightarrow C = q \overline{\dots} a$ thus $u \upharpoonright B \rightarrow C$ is complete. Suppose that $u \upharpoonright A \rightarrow B$ is not complete, then its first move is unanswered, but since this is a B -move, it must also occur unanswered in $u \upharpoonright B \rightarrow C$ which is a contradiction since we have just prove that $u \upharpoonright B \rightarrow C$ is complete. Thus $u \upharpoonright A \rightarrow B$ is also complete.

The induction hypothesis permits us to conclude. \square

Consequently if $u \upharpoonright \text{type}(T)$ is complete then u is maximal: no move (and in particular no internal move) can be played after u .

4.2.1.3 Fully-revealed and syntactically-revealed semantics

We call *revealed semantics* any game model of a language in which a term is denoted by some revealed strategy as defined in the previous section. As we have already observed, depending on the internal moves that we wish to hide, we obtain different possible revealed strategies for a given term. Hence there is not a unique way to define a revealed semantics. In this section we give two examples of such semantics.

Let π_i denote the i^{th} projection strategy $\pi_i : \llbracket X_1 \times \dots \times X_l \rrbracket \rightarrow \llbracket X_i \rrbracket$.

Definition 4.2.7 (The fully-revealed semantics). The **fully-revealed game denotation** of M written $\langle\langle \Gamma \vdash M : A \rangle\rangle$ is defined by structural induction on the η -long normal form of M :

$$\begin{aligned} \langle\langle \Gamma \vdash \alpha : o \rangle\rangle &= \llbracket \Gamma \vdash \alpha : o \rrbracket \quad \text{where } \alpha \in \Gamma \cup \Sigma, \\ \langle\langle \Gamma \vdash \lambda \bar{\xi}. M : A \rangle\rangle &= \Lambda^{\bar{\xi}}(\langle\langle \Gamma, \bar{\xi} \vdash M : o \rangle\rangle) \\ \langle\langle \Gamma \vdash x_i N_1 \dots N_p : o \rangle\rangle &= \langle \pi_i, \langle\langle \Gamma \vdash N_1 : A_1 \rangle\rangle, \dots, \langle\langle \Gamma \vdash N_p : A_p \rangle\rangle \rangle \parallel ev^p, \quad X_i = A_0 \\ \langle\langle \Gamma \vdash f N_1 \dots N_p : o \rangle\rangle &= \langle \langle\langle \Gamma \vdash N_1 : A_1 \rangle\rangle, \dots, \langle\langle \Gamma \vdash N_p : A_p \rangle\rangle \rangle \parallel \llbracket f \rrbracket, \quad f : A_0 \in \Sigma \\ \langle\langle \Gamma \vdash N_0 \dots N_p : o \rangle\rangle &= \langle \langle\langle \Gamma \vdash N_0 : A_0 \rangle\rangle, \dots, \langle\langle \Gamma \vdash N_p : A_p \rangle\rangle \rangle \parallel ev^p \end{aligned}$$

where $\Gamma = x_1 : X_1 \dots x_l : X_l$, $A_0 = (A_1, \dots, A_p, o)$ and ev^p denotes the evaluation strategy with p parameters where $p \geq 1$.

Fig. 4.1 shows tree representations of the interaction games involved in the revealed strategy $\langle\langle \Gamma \vdash M : A \rangle\rangle$ for the two application cases. These two trees give us information about the constituent strategies involved in $\langle\langle M \rangle\rangle$. For instance the revealed strategy $\langle\langle N_0 \rangle\rangle$ is defined on the interaction game $\langle\langle T^{00} \rangle\rangle$ whose root game is $A \rightarrow B_0$, and the strategy ev is defined on the interaction game $\langle\langle T^1 \rangle\rangle$ whose underlying tree is constituted of a single game-node $B_0 \times \dots \times B_p \rightarrow o$ (thus plays of ev do not contain any uncovered move).

Example 4.2.2. Take the term $\lambda x.(\lambda f.f x)(\lambda y.y)$. Its fully-revealed denotation is

$$\Lambda(\llbracket x : X \vdash \lambda f.f x : (o \rightarrow o) \rightarrow o \rrbracket, \llbracket x : X \vdash \lambda y.y : o \rightarrow o \rrbracket \parallel ev^2) .$$

Note that the set of fully-revealed strategy is not a category (composition is not associative and there is no identity strategy).

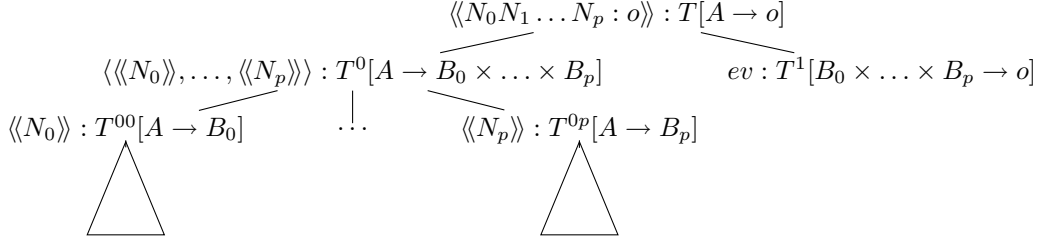
Definition 4.2.8 (Syntactically-revealed semantics). The **syntactically-revealed game denotation** of M written $\langle\langle \Gamma \vdash M : A \rangle\rangle_s$ is defined by structural induction on the η -long normal form of M . The equations are the same as in Definition 4.2.7 except for the third case:

$$\langle\langle \Gamma \vdash x_i N_1 \dots N_p : o \rangle\rangle_s = \langle \pi_i, \langle\langle \Gamma \vdash N_1 : A_1 \rangle\rangle_s, \dots, \langle\langle \Gamma \vdash N_p : A_p \rangle\rangle_s \rangle;^{\emptyset, \{1..p\}} ev^p, \quad X_i = A_0 .$$

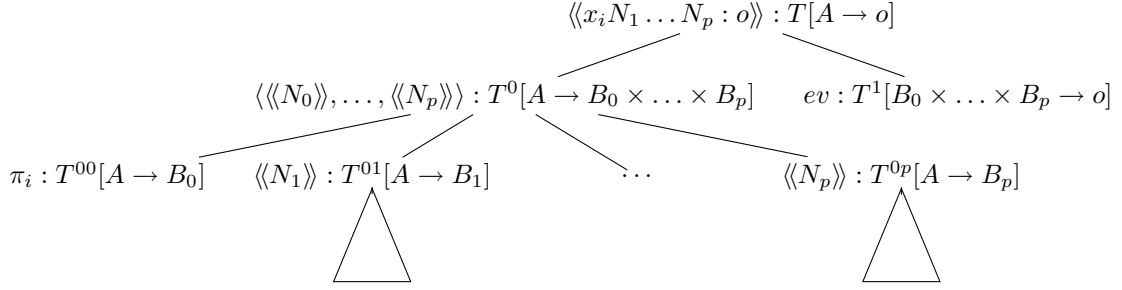
The syntactically-revealed denotation differs from the fully-revealed one in that only certain internal moves are preserved during composition: when computing the denotation of an application joint by an @-node in the computation tree, all the internal moves are preserved; when computing the denotation of $\langle\langle y_i N_1 \dots N_p \rangle\rangle$ for some variable y_i , however, we only preserve the internal moves of N_1, \dots, N_p while omitting the internal moves produced by the copy-cat projection strategy denoting y_i .

4.2.1.4 Relating the two revealed denotations

As one would expect, the two revealed denotations that we have just introduced are in fact isomorphic: there is a transformation that maps $\langle\langle \Gamma \vdash M : A \rangle\rangle$ to $\langle\langle \Gamma \vdash M : A \rangle\rangle_s$ and another one mapping $\langle\langle \Gamma \vdash M : A \rangle\rangle_s$ to $\langle\langle \Gamma \vdash M : A \rangle\rangle$. That is what we are about to show in this section.



Tree-representation of the revealed strategy $\langle\langle \Gamma \vdash N_0 N_1 \dots N_p : o \rangle\rangle$.



Tree-representation of the revealed strategy $\langle\langle \overline{x} : \overline{X} \vdash x_i N_1 \dots N_p : o \rangle\rangle$.

A node label ' $\Pi : T[G]$ ' indicates that Π is a revealed strategy on the interaction game T whose top-level game (at the root of the tree underlying T) is G . Each game is annotated with a string $s \in \{0..p\}^*$ in the exponent to indicate the path from the root to the corresponding node in the tree. (The digits in s tell the direction to take at each branch of the tree.)

The games A and B are given by:

$$\begin{aligned} A &= X_1 \times \dots \times X_n \\ B &= \underbrace{((B'_1 \times \dots \times B'_p) \rightarrow o')}_{B_0} \times B_1 \times \dots \times B_p . \end{aligned}$$

Figure 4.1: Tree-representation of the revealed strategy in the application case.

The two revealed denotations are isomorphic Take a term M of the form $x_i N_1 \dots N_p$ (This is the only case where the two revealed denotations differ). We use the notations introduced in Fig. 4.1. We have $\langle\langle \Gamma \vdash M : o \rangle\rangle = \Sigma \parallel ev$ and $\langle\langle \Gamma \vdash M : o \rangle\rangle_s = \Sigma_s;^{\emptyset, \{1..p\}} ev$ where $\Sigma = \langle \pi_i, \langle\langle \Gamma \vdash N_1 : B_1 \rangle\rangle, \dots, \langle\langle \Gamma \vdash N_p : B_p \rangle\rangle \rangle$ and $\Sigma_s = \langle \pi_i, \langle\langle \Gamma \vdash N_1 : B_1 \rangle\rangle_s, \dots, \langle\langle \Gamma \vdash N_p : B_p \rangle\rangle_s \rangle$. For both denotations, the composition takes place on the game

$$X_1 \times \dots \times \overbrace{((B_1'' \times \dots \times B_p'') \rightarrow o'')}^{X_i} \times X_n \xrightarrow{\Sigma} \boxed{\overbrace{((B_1' \times \dots \times B_p') \rightarrow o')}^{B_0} \times B_1 \times \dots \times B_p} \xrightarrow{ev} o$$

where the dashed-line frame contains the internal components of the game. We will represent the superficial internal moves using the symbols \bullet , \circ , \bullet , \circ for OP-move, PO-move, O-move and P-move respectively.

In $\langle\langle \Gamma \vdash M : o \rangle\rangle$, all the internal moves (including the profound ones) from B_k for $k \in \{0..p\}$ are preserved, whereas in $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$, the internal moves from B_0 as well as the superficial internal moves in B_k for $k \in \{1..p\}$ are hidden.

Clearly, $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ can be obtained from $\Sigma_s \parallel ev$ and conversely. Every play u of $\Sigma_s \parallel ev$ corresponds to the play of $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ obtained by removing all the superficial B_k -moves from u ; and for any $u \in \langle\langle \Gamma \vdash M : o \rangle\rangle_s$, there is a unique play v of $\Sigma_s \parallel ev$ ending with an external move such that we obtain u back after removing the superficial internal moves from v . Hence by repeating the same argument recursively at every subterm of M of the form $y_i N_1 \dots N_p$, we obtain that there exists a reversible transformation mapping $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ to $\langle\langle \Gamma \vdash M : o \rangle\rangle$.

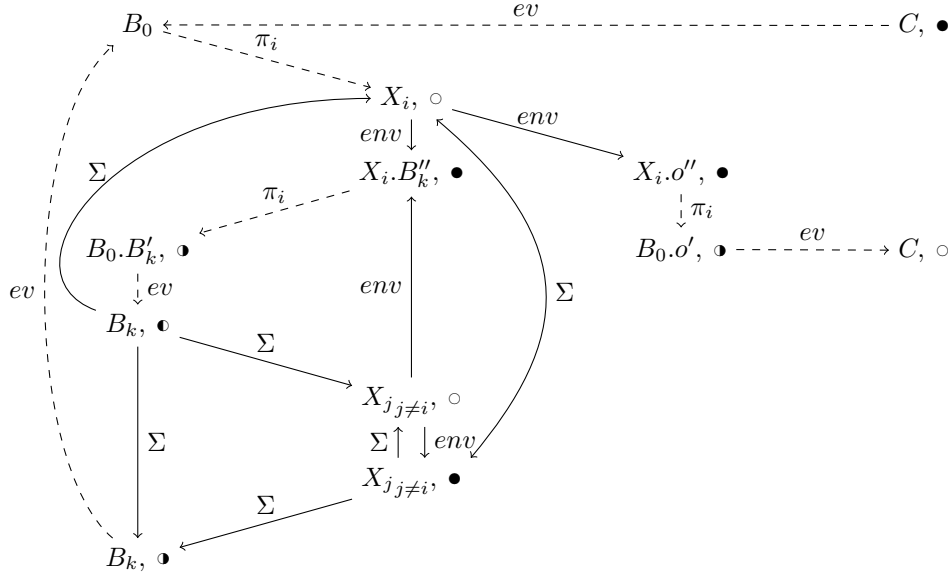
Recovering the fully-revealed semantics from the syntactically-revealed semantics

We now explicitly define the transformation that turns $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ into $\langle\langle \Gamma \vdash M : o \rangle\rangle$. To achieve this, it suffices to define constructively the function mentioned in the previous paragraph which given an interaction play u of $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ returns v , the unique play of $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ ending with an external move such that removing the superficial internal moves from v gives us back u .

We analyse the structure of an interaction play of $\Sigma \parallel ev$. The diagram in Fig. 4.2 gives us a precise description of the flow of an interaction play. The nodes of the diagram represent the moves. A node label is of the form A, α where $\alpha \in \{\bullet, \circ, \bullet, \circ\}$: the first component is the game in which the move is played and the second indicates which player is making the move. The notation $X_i.B_k''$ is used to denote the sub-component B_k'' of X_i . The edges indicate the moves that are allowed in a given state of the game. The game starts at node C, \bullet which correspond to the initial move of the overall game. Each edge is labelled with the strategy that plays the corresponding move. For instance the edge $B_k, \circ \xrightarrow{ev} B_0, \bullet$ tells us that if B_k, \circ is the last move played then the evaluation strategy allows the other user to respond with the move B_k, \bullet . The copy-cat strategies are highlighted with dashed-edges.

We observe that every (superficial) internal move played in some component B_k for $k \in \{0..p\}$ is either a copy of a previous external move, or it is subsequently copied to an external component by the copy-cat strategy ev or π_i . For instance, the \bullet -moves from B_0 are copies by ev of O-moves from C and \circ -moves from B_k for $k \in \{1..p\}$; the \circ -moves from B_0 are copies by π_i of O-move from X_i ; and \bullet -moves from B_k for every $k \in \{1..p\}$ are copies by ev of \circ -moves from the components B_k' of B_0 .

Moreover, each move on the diagram of Fig. 4.2 has either a single outgoing copy-cat edge in which case the following move is uniquely determined, or it has multiple out-going edges all labelled by Σ in which case the strategy Σ determines which moves will be played next. Hence for any two consecutive move in a play of $\langle\langle \Gamma \vdash M : o \rangle\rangle_s$ we can uniquely recover all the internal moves occurring between the two moves in the corresponding play of $\langle\langle \Gamma \vdash M : o \rangle\rangle$ by following the arrows of the flow diagram.



where $k \in \{1..p\}$ and $i, j \in \{1..n\}$.

Figure 4.2: Flow-diagram for interaction plays of $\langle\langle \Gamma \vdash x_i N_1 \dots N_p \rangle\rangle$.

4.2.1.5 Revealed semantics versus standard game semantics

In the standard semantics, given two strategies $\sigma : A \rightarrow B$, $\tau : B \rightarrow C$ and a sequence $s \in \sigma; \tau$, it is possible to (uniquely) recover the internal moves from the sequence s that have been hidden during composition [HO00, part II]. The revealed denotation of a term can be recovered from its standard game denotation by recursively uncovering the internal moves for every application occurring in the term.

Conversely, the standard denotation can be obtained from the revealed denotation by filtering out all the internal moves:

$$\llbracket \Gamma \vdash M : T \rrbracket = \langle\langle \Gamma \vdash M : T \rangle\rangle \upharpoonright \llbracket \Gamma \rightarrow T \rrbracket . \quad (4.7)$$

This equality remains valid if we replace the fully revealed denotation by the syntactically revealed denotation.

Observe that the two set of plays $\langle\langle \Gamma \vdash M : T \rangle\rangle$ and $\llbracket \Gamma \vdash M : T \rrbracket$ are not in bijection. Indeed, by definition the revealed denotation is prefix-closed therefore it also contains plays ending with an internal move. Thus the revealed denotation contains more plays than the standard denotation.

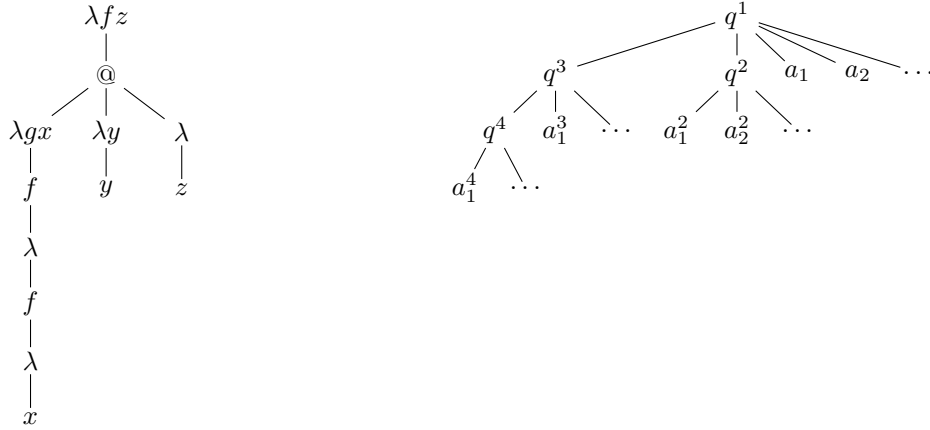
In fact, the set of plays $\llbracket \Gamma \vdash M : T \rrbracket$ is in bijection with the subset of $\langle\langle \Gamma \vdash M : T \rangle\rangle$ consisting of plays ending with an external move. Additionally, the set of complete plays of $\llbracket \Gamma \vdash M : T \rrbracket$ is in bijection with the set of complete interaction plays of $\langle\langle \Gamma \vdash M : T \rangle\rangle$.

4.2.2 Relating computation trees and games

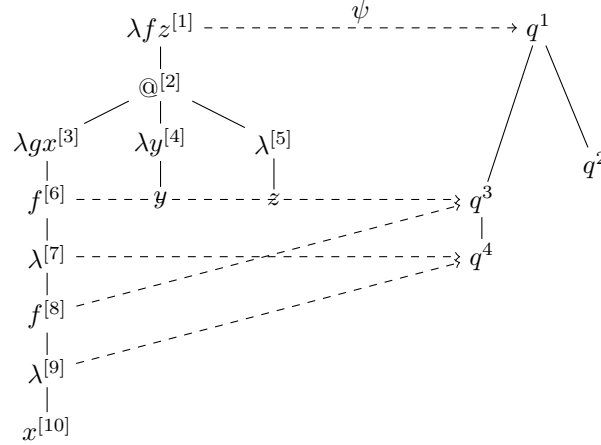
In this paragraph we are interested in showing a relation between nodes of the computation tree and moves of the game arena. We start first with an example to get the insight before giving the formal definition.

4.2.2.1 Example

Consider the following term $M \equiv \lambda f z. (\lambda g x. f(fx)) (\lambda y. y) z$ of type $(o \rightarrow o) \rightarrow o \rightarrow o$. Its η -long normal form is $\lambda f z. (\lambda g x. f(fx)) (\lambda y. y) (\lambda. z)$. The following figure represents the computation tree of M on the left and the arena of the game $\llbracket (o \rightarrow o) \rightarrow o \rightarrow o \rrbracket$ on the right:



The figure below represents the computation tree (left) and the arena (right). The dashed line defines a partial function ψ from the set of nodes in the computation tree to the set of moves. For simplicity, we now omit answer moves when representing arenas.



Consider the justified sequence of moves $s \in \llbracket M \rrbracket$:

$$s = q^1 \overset{\curvearrowright}{q^3} \overset{\curvearrowright}{q^4} \overset{\curvearrowright}{q^3} \overset{\curvearrowright}{q^4} q^2 \in \llbracket M \rrbracket .$$

There is a corresponding justified sequence of nodes in the computation tree:

$$r = \lambda f z . f^{[6]} . \lambda^{[7]} . f^{[8]} . \lambda^{[9]} . z$$

such that $s_i = \psi(r_i)$ for all $i < |s|$.

The sequence r is in fact the reduction of the following traversal:

$$t = \lambda f z . @^{[2]} . \lambda g x^{[3]} . f^{[6]} . \lambda^{[7]} . f^{[8]} . \lambda^{[9]} . x^{[10]} . \lambda^{[5]} . z .$$

By representing side-by-side the computation tree and the type arena of a term in η -normal form we have observed that some nodes of the computation tree can be mapped to question moves of the arena. In the next section, we show how to define this mapping in a systematic manner.

4.2.2.2 Formal definition

We now establish formally the relationship between games and computation trees. We assume that a term $\Gamma \vdash M : T$ in η -long normal form is given.

NOTATIONS 4.2.1 We suppose that computation tree $\tau(M)$ is given by a pair (V, E) where V is the set of vertices and $E \subseteq V \times V$ is the parent-child relation. We have $V = N \cup L$ where N and L are the set of nodes and value-leaves respectively. Let \mathcal{D} be the set of values of the base type o . If n is a node in N then the value-leaves attached to the node n are written v_n where v ranges in \mathcal{D} . Similarly, if q is a question in A then the answer moves enabled by q are written v_q where v ranges in \mathcal{D} .

Definition 4.2.9 (Mapping from nodes to moves of the standard game-semantics).

- Let n be a node in $N_\lambda \cup N_{\text{var}}$ and q be a question move of some game A such that n and q are of type (A_1, \dots, A_p, o) for some $p \geq 0$. Let $\{q^1, \dots, q^p\}$ (resp. $\{v_q \mid v \in \mathcal{D}\}$) be the set of question (resp. answer) moves enabled by q in A (each q^i being of type A_i).

We define the function $\psi_A^{n,q}$ from V^{n^\vdash} (the nodes from V that are hereditarily enabled by n) to moves of A as:

$$\begin{aligned} \psi_A^{n,q} &= \{n \mapsto q\} \cup \{v_n \mapsto v_q \mid v \in \mathcal{D}\} \\ &\cup \begin{cases} \bigcup_{m \in N_{\text{var}} \mid n \vdash_i m} \psi_A^{m,q^i}, & \text{if } n \in N_\lambda ; \\ \bigcup_{i=1..p} \psi_A^{n,i,q^i}, & \text{if } n \in N_{\text{var}} . \end{cases} \end{aligned}$$

- Suppose $\Gamma = x_1 : X_1, \dots, x_k : X_k$. Let q_0 denote $\llbracket \Gamma \rightarrow T \rrbracket$'s initial move³ and suppose that the set of moves enabled by q_0 in $\llbracket \Gamma \rightarrow T \rrbracket$ is $\{q_{x_1}, \dots, q_{x_k}, q^1, \dots, q^p\} \cup \{v_q \mid v \in \mathcal{D}\}$ where each q^i is of type A_i and q_{x_j} of type X_j .

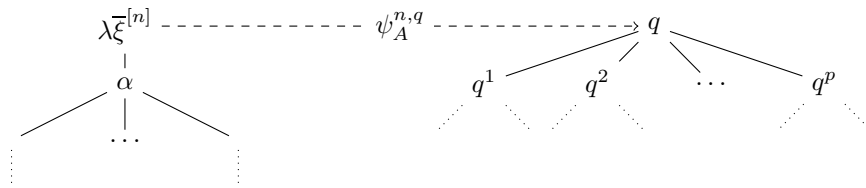
We define $\psi_M : V^{\otimes \vdash} \rightarrow \llbracket \Gamma \rightarrow T \rrbracket$ (or just ψ if there is no ambiguity) as:

$$\begin{aligned} \psi_M &= \{r \mapsto q_0\} \cup \{v_r \mapsto v_{q_0} \mid v \in \mathcal{D}\} \\ &\cup \bigcup_{n \in N_{\text{var}} \mid \otimes \vdash_i n} \psi_{\llbracket \Gamma \rightarrow T \rrbracket}^{n,q^i} \\ &\cup \bigcup_{n \in N_{\text{iv}} \mid n \text{ labelled } x_j, j \in \{1..k\}} \psi_{\llbracket \Gamma \rightarrow T \rrbracket}^{n,q_{x_j}} . \end{aligned}$$

It can easily be checked that the domain of definition of $\psi_A^{n,q}$ is indeed the set of nodes that are hereditarily enabled by n and similarly, the domain of ψ_M is the set of nodes that are hereditarily enabled by the root (this includes free variable nodes and nodes that are hereditarily enabled by free variable nodes). Also, if M is closed then we have $\psi_M = \psi_{\llbracket \rightarrow T \rrbracket}^{\otimes, q_0}$.

The construction of the function $\psi_A^{n,q}$, defined above, goes as follows. Let p be the arity of the type of n and q .

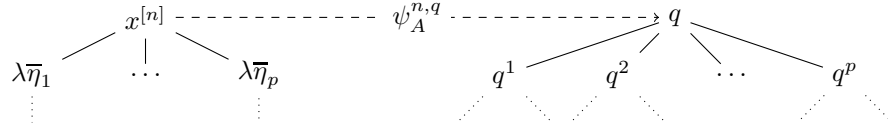
- If $p = 0$ then n is a dummy λ -node or a ground type variable: $\psi_A^{n,q}$ maps n to the initial move q .
- If $p \geq 1$ and $n \in N_\lambda$ with n labelled $\lambda \bar{\xi} = \lambda \xi_1 \dots \xi_p$ then the sub-computation tree rooted at n and the arena A have the following forms (value-leaves and answer moves are not represented for simplicity):



³Arenas involved in the game semantics of simply typed lambda-calculus are all trees: they have a single initial move.

For each abstracted variable ξ_i there exists a corresponding question move q^i of the same order in the arena. $\psi_A^{n,q}$ maps each free occurrence of ξ_i in the computation tree to the move q^i .

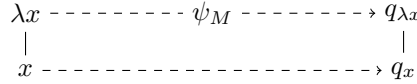
- If $p \geq 1$ and $n \in N_{\text{var}}$ then n is labelled with a variable $x : (A_1, \dots, A_p, o)$ with children nodes $\lambda\bar{\eta}_1, \dots, \lambda\bar{\eta}_p$. The computation tree $\tau(M)$ rooted at n and the arena A have the following forms:



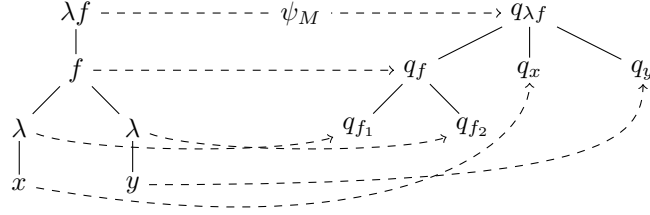
and $\psi_A^{n,q}$ maps each node $\lambda\bar{\eta}_i$ to the question move q^i .

Example 4.2.3. For each of the following examples of term $\Gamma \vdash M : T$, we represent the computation tree $\tau(M)$, the arena of the game $\llbracket \Gamma \rightarrow T \rrbracket$, and the function ψ_M (in dashed lines):

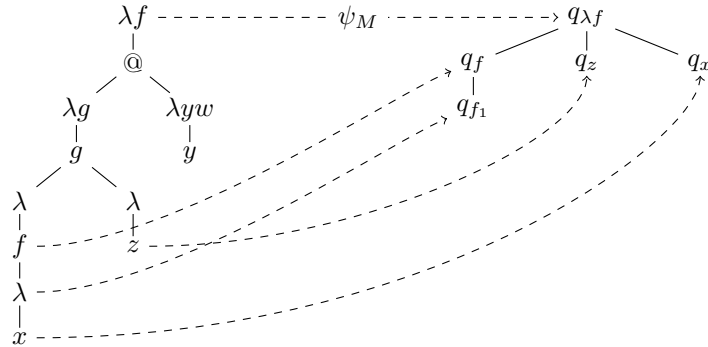
- $M = \lambda x^o.x$



- $M = \lambda f^{(o,o,o)}.fxy$



- $M = \lambda f^{(o,o)} . (\lambda g^{(o,o,o)} . g(fx)z) (\lambda y^o w^o . y)$



Lemma 4.2.2.

- (i) ψ_M maps λ -nodes to O -questions, variable nodes to P -questions, value-leaves of λ -nodes to P -answers and value-leaves of variable nodes to O -answers;
- (ii) ψ_M preserves hereditary enabling: a node $n \in V^{\oplus \vdash}$ is hereditarily enabled by some node $n' \in V^{\oplus \vdash}$ in $\tau(M)$ if and only if the move $\psi_M(n)$ is hereditarily enabled by $\psi_M(n')$ in $\llbracket \Gamma \rightarrow T \rrbracket$;
- (iii) ψ_M maps a node of a given order to a move of the same order;
- (iv) Let $s \in \text{Trav}(M)^{\dagger \oplus}$. The P -view (resp. O -view) of $\psi_M(s)$ and s are computed identically (i.e., the set of positions of occurrences that need to be deleted from each sequences in order to obtain their respective P -view (resp. O -view) is the same for both sequences).

Proof. (i), (ii) and (iii) are direct consequences of the definition.

(iv) Because of (i) and since t and $\psi_M(t)$ have the same pointers, the computations of the P-view (resp. O-view) of the sequence of moves and the P-view (resp. O-view) of the sequence of nodes follow the same steps. \square

The convention chosen to define the order of the root node (see Def. 4.1.3) permits us to have property (iii). This explains why the order of the root node and other lambda nodes were defined differently.

By extension, we can define the function ψ_M on $\mathcal{Trav}(M)^{\text{!}\otimes}$, the set of traversal reductions, as follows:

Definition 4.2.10 (Mapping sequences of nodes to sequences of moves). The function ψ_M maps any traversal reduction $u = u_0 u_1 \dots \in \mathcal{Trav}(M)^{\text{!}\otimes}$ to the following justified sequence of moves of the arena $\llbracket \Gamma \rightarrow T \rrbracket$: $\psi_M(u) = \psi_M(u_0) \psi_M(u_1) \psi_M(u_2) \dots$ where $\psi_M(u)$ is equipped with u 's pointers.

The pointer-free function underlying ψ_M is thus a monoid homomorphism.

4.2.3 Mapping traversals to interaction plays

Let I be the interaction game of the revealed strategy $\langle\langle \Gamma \vdash M : T \rangle\rangle_s$ and M_I be the set of equivalence class of moves from \mathcal{M}_I .

Let r' be a lambda node in N_{spawn} . We write $\Gamma(r') \vdash \kappa(r') : T(r')$ to denote the subterm of $[M]$ rooted at r' (thus $\Gamma(r') \subseteq \Gamma$). We consider the function $\psi_{\kappa(r')}$ which maps nodes of $V^{r' \vdash}$ to moves of $\llbracket \Gamma(r') \rightarrow T(r') \rrbracket$. Since \mathcal{M}_I contains the moves from the standard game $\llbracket \Gamma(r') \rightarrow A(r') \rrbracket$, we can consider $\psi_{\kappa(r')}$ as a function from $V^{r' \vdash}$ to \mathcal{M}_I .

Every node in $n \in V \setminus (V_{\otimes} \cup V_{\Sigma})$ is either hereditarily enabled by the root or by some λ -node in N_{spawn} (the children nodes of \otimes/Σ -nodes). Therefore we can define the following relation ψ_M^* from $V \setminus (V_{\otimes} \cup V_{\Sigma})$ to \mathcal{M}_I :

$$\psi_M^* = \psi_M \cup \bigcup_{r' \in N_{\text{spawn}}} \psi_{\kappa(r')} .$$

This relation is totally defined on $V \setminus (V_{\otimes} \cup V_{\Sigma})$ since those nodes are either hereditarily justified by the root, by an \otimes -node or by a Σ -node. Moreover it is a relation and *not* a function since for a given variable node x , for every spawn node r' occurring in the path from x to \otimes , x is hereditarily enabled by r' with respect to the computation tree $\tau(\kappa(r'))$. Thus the domains of definition of the relations $\psi_{\kappa(r')}$ for such nodes r' overlap. It can be easily check, however, that for any node $n \in V \setminus (V_{\otimes} \cup V_{\Sigma})$, the moves in $\psi_M^*(n)$ are all \sim -equivalent, which leads us to the following definition:

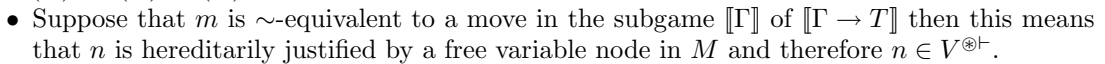
Definition 4.2.11 (Mapping from nodes to moves of the syntactically-revealed semantics). We define the function $\varphi_M : V \setminus (V_{\otimes} \cup V_{\Sigma}) \rightarrow M_I$ as follows: for $n \in V \setminus (V_{\otimes} \cup V_{\Sigma})$, $\varphi_M(n)$ is defined as the \sim -equivalence class containing the set $\psi_M^*(n)$. We omit the subscript in φ_M if there is no ambiguity.

Definition 4.2.12 (Mapping sequences of nodes to sequences of moves). We define the function φ_M from $\mathcal{Trav}(M)^*$ to justified sequence of moves in M_I as follows. If $u = u_0 u_1 \dots \in \mathcal{Trav}(M)^*$ then:

$$\varphi_M(s) = \varphi_M(u_0) \varphi_M(u_1) \varphi_M(u_2) \dots$$

where $\varphi_M(u)$ is equipped with u 's pointers.

Example 4.2.4. Take $M = \lambda x^o. (\lambda g^{(o,o)}. g x z) (\lambda y^o. y)$. The diagram below represents the computation tree (middle) and the relation $\psi_M^* = \psi_{\lambda x} \cup \psi_{\lambda g.gx} \cup \psi_{\lambda y.y}$ (dashed-lines).



- Suppose that m is \sim -equivalent to a move in the subgame $\llbracket T \rrbracket$ of $\llbracket \Gamma \rightarrow T \rrbracket$ then m necessarily belongs to the subgame $\Gamma(\odot)$ of $\llbracket \Gamma(\odot) \rightarrow T(\odot) \rrbracket$. Indeed, since \odot 's parent node is an application node, moves in the subgame $\llbracket T(\odot) \rrbracket$ correspond to internal moves of the application. By definition of the interaction strategy for the application case, such moves can only be \sim -equivalent to other internal moves and thus cannot be equivalent to move in $\llbracket T \rrbracket$. Consequently, n is her. just. by a free variable node z in $\kappa(\odot)$. By assumption, \odot is the closest node to the root r (excluding r itself) for which n belongs to $V^{\odot \vdash}$ (the domain of definition of $\psi_{\kappa(\odot)}$). Hence z is not bound by any λ -node occurring in the path to the root. Thus $z \in V^{\oplus \vdash}$ and therefore $n \in V^{\oplus \vdash}$.

Hence $H = V^{\oplus \vdash}$. Consequently, for any traversal t of M we have $\varphi_M(t^*) \upharpoonright \llbracket \Gamma \rightarrow T \rrbracket = \varphi_M(t^* \upharpoonright V^{\oplus \vdash})$ which is equal to $\varphi_M(t \upharpoonright r)$ by Lemma 4.1.10. \square

4.2.4 The correspondence theorem for the pure simply typed lambda calculus

In this section, we establish a connection between the interaction semantics of a simply typed term without interpreted constants (*i.e.*, $\Sigma = \emptyset$) and the traversals of its computation tree: we show that the set $\mathcal{Trav}(M)$ of traversals of the computation tree is isomorphic to the set of uncovered plays of the strategy denotation (this is the counterpart of Ong's "Path-Traversal Correspondence" Theorem [Ong06a]), and that the set of traversal reductions is isomorphic to the strategy denotation.

Preliminary lemmas

NOTATION 4.2.2 For any node occurrence n in a justified sequence (of nodes or of moves) u we write $\text{ptrdist}_u(n)$, or just $\text{ptrdist}(n)$ if there is no ambiguity, to denote the distance between n and its justifier in u if it has one, and 0 otherwise.

Lemma 4.2.4.

$$\left(\begin{array}{l} t \cdot n_1, t \cdot n_2 \in \mathcal{Trav}(M) \\ \wedge n_1 \neq n_2 \end{array} \right) \implies n_1, n_2 \in V_{\lambda}^{\oplus \vdash} \wedge (\psi(n_1) \neq \psi(n_2) \vee \text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)) .$$

Proof. Take $t \cdot n_1, t \cdot n_2 \in \mathcal{Trav}(M)$. Suppose that n_1 and n_2 belong to two distinct categories of node (N_{var} , N_{\odot} , N_{λ} , N_{Σ} , L_{var} , L_{\odot} , or L_{λ} , L_{Σ}) then necessarily one must be visited with the rule (InputVar) and the other by (InputVal) (they are the only rules that have a common domain of definition), thus one is a leaf-node and the other is an inner node which implies that $\psi(n_1) \neq \psi(n_2)$.

Otherwise n_1 and n_2 belong to the same category of nodes and we proceed by case analysis:

- If $n_1, n_2 \in N_{\odot}$ then $t \cdot n_1$ and $t \cdot n_2$ are formed using the (App) rule. Since this rule is deterministic we must have $n_1 = n_2$ which violates the second hypothesis.
- If $n_1, n_2 \in L_{\odot}$ then the traversals are formed using the deterministic rule ($\text{Value}^{\odot \mapsto \lambda}$) which again violates the second hypothesis.
- If $n_1, n_2 \in N_{\Sigma}$ then they are formed using a deterministic constant rule (see Def. 4.1.13).
- If $n_1, n_2 \in L_{\Sigma}$ then they are formed using a deterministic value-constant rule.
- If $n_1, n_2 \in N_{\text{var}}$ then $t \cdot n_1$ and $t \cdot n_2$ were formed using either rule (Lam) or (App). But these two rules are deterministic and their domains of definition are disjoint. Hence again the second hypothesis is violated.
- If $n_1, n_2 \in L_{\text{var}}$ then either the traversals were both formed using the deterministic rule ($\text{Value}^{\text{var} \mapsto \lambda}$) in which case the second hypothesis is violated; or they were formed with (InputValue) in which case n_1 and n_2 are two different value leaves belonging to $V_{\lambda}^{\oplus \vdash}$ and justified by the same input variable node. Thus by definition of ψ , $\psi(n_1) \neq \psi(n_2)$.
- If $n_1, n_2 \in N_{\lambda}$ then the traversals $t \cdot n_1$ and $t \cdot n_2$ must have been formed using either rule (Root), (App), (Var) or (InputVar). Since all these rules have disjoint domains of definition, the same rule must have been used to form $t \cdot n_1$ and $t \cdot n_2$. But since the rules (Root), (App) and (Var) are all deterministic, the rule used is necessarily (InputVar).

By definition of (InputVar), $n_1, n_2 \in N_{\lambda}^{\oplus \vdash}$ and the parent node of n_1 and the parent node of n_2 occur in $\sqsubseteq t_{\leq x}$ where $x \in N_{\text{var}}^{\oplus \vdash}$ denotes the pending node at t . If n_1 and n_2 have the same parent node in $\tau(M)$ then since $n_1 \neq n_2$, by definition of ψ , $\psi(n_1) \neq \psi(n_2)$. If their parent node is different, then n_1 and n_2 are necessarily justified by two different occurrences in t therefore $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$.

- If $n_1, n_2 \in L_{\lambda}$ then either the traversals $t \cdot n_1$ and $t \cdot n_2$ were formed using $(\text{Value}^{\lambda \mapsto \text{var}})$ or they were formed with $(\text{Value}^{\lambda \mapsto \oplus})$ but this is impossible since these two rules are deterministic and $n_1 \neq n_2$. \square

The function φ_M regarded as a function from the set of vertices $V \setminus V_{\oplus}$ of the computation tree to moves in arenas is not injective. (For instance the two occurrences of x in the computation tree of $\lambda f x. f x x$ are mapped to the same question move.) However the function φ_M defined on the set of \oplus -free traversals is injective, and similarly the function ψ_M defined on the set of traversal reduction is injective:

Lemma 4.2.5 (ψ_M and φ_M are injective). *For any two traversals t_1 and t_2 :*

- (i) *If $\varphi(t_1^*) = \varphi(t_2^*)$ then $t_1^* = t_2^*$;*
- (ii) *if $\psi(t_1 \upharpoonright \oplus) = \psi(t_2 \upharpoonright \oplus)$ then $t_1 \upharpoonright \oplus = t_2 \upharpoonright \oplus$.*

Proof. (i) The result is trivial if either t_1 or t_2 is empty. Otherwise, suppose that $t_1^* \neq t_2^*$ then necessarily $t_1 \neq t_2$. W.l.o.g. we can assume that the two traversals differ only by their last node (or last node's pointer). Thus we have $t_1 = t \cdot n_1$ and $t_2 = t \cdot n_2$ for some sequence t and some occurrences n_1, n_2 where either n_1 and n_2 are two distinct node in the computation tree or $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$.

If $n_1 = n_2$ and $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$ then n_1, n_2 are not \oplus -nodes nor Σ -nodes (since for such nodes we would have $\text{ptrdist}(n_1) = 0 = \text{ptrdist}(n_2)$). By definition of the sequence $\varphi(t_1)$ we have $\text{ptrdist}(\varphi(n_1)) = \text{ptrdist}(n_1)$ and similarly $\text{ptrdist}(\varphi(n_2)) = \text{ptrdist}(n_2)$ thus $\varphi(t' \cdot n_1) \neq \varphi(t' \cdot n_2)$. Finally since $n_1, n_2 \notin (N_{\oplus} \cup N_{\Sigma})$ we also have $\varphi((t' \cdot n_1)^*) \neq \varphi((t' \cdot n_2)^*)$. Hence $\varphi(t_1^*) \neq \varphi(t_2^*)$.

If $n_1 \neq n_2$ then by Lemma 4.2.4 n_1, n_2 are not \oplus -nodes or Σ -nodes (since such nodes are not her. just. by the root) and we have either $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$ or $\varphi(n_1) = \psi(n_1) \neq \psi(n_2) = \varphi(n_2)$. Clearly both cases imply $\varphi(t_1^*) \neq \varphi(t_2^*)$.

(ii) Suppose that $t_1 \upharpoonright \oplus \neq t_2 \upharpoonright \oplus$ then necessarily $t_1 \neq t_2$. W.l.o.g. we can assume that the two sequences differ only by their last occurrence. Hence we have $t_1 = t \cdot n_1$, $t_2 = t' \cdot n_2$ for some sequence t and some nodes n_1, n_2 where either $n_1 \neq n_2$ or $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$.

If $n_1 \neq n_2$ then Lemma 4.2.4 gives $\psi(t_1 \upharpoonright \oplus) \neq \psi(t_2 \upharpoonright \oplus)$.

If $n_1 = n_2$ then $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$. (InputVar) and (InputValue) are the only rules that can visit the same node with two different pointers, thus n_1 and n_2 must be in $V_{\lambda}^{\oplus \vdash}$. Hence:

$$\psi(t_i \upharpoonright \oplus) = \psi(t \upharpoonright \oplus) \cdot \psi(n_i) \text{ for } i \in \{1..2\}$$

where $\text{ptrdist}_{\psi(t_i \upharpoonright \oplus)}(\psi(n_i)) = \text{ptrdist}_{t_i \upharpoonright \oplus}(n_i)$.

Furthermore, since $\text{ptrdist}(n_1) \neq \text{ptrdist}(n_2)$ and $t_1 \upharpoonright_{< n_1} = t_2 \upharpoonright_{< n_2}$ we have $\text{ptrdist}_{t_1 \upharpoonright \oplus}(n_1) \neq \text{ptrdist}_{t_2 \upharpoonright \oplus}(n_2)$. Thus $\psi(t_1 \upharpoonright \oplus) \neq \psi(t_2 \upharpoonright \oplus)$. \square

Corollary 4.2.1.

- (i) *φ defines a bijection from $\text{Trav}(M)^*$ to $\varphi(\text{Trav}(M)^*)$;*
- (ii) *ψ defines a bijection from $\text{Trav}(M)^{\upharpoonright \oplus}$ to $\psi(\text{Trav}(M)^{\upharpoonright \oplus})$.*

The following lemma says that extending a traversal locally also extends the traversal globally: the traversal t of M can be extended by extending a sub-traversal t' of some subterm of M . This is not obvious since t' is a subsequence of t which means that the nodes in t' are also present in t with the same pointers but with some other nodes interleaved in between. However these interleaved nodes are inserted in a preservative way which allows us to apply on t the rule that was used to extend sub-traversal t' :

Lemma 4.2.6 (Sub-traversal progression). *Let \otimes_j be a lambda node in $\tau(M)$, t be a justified sequence of nodes of $\tau(M)$, and r_j be an occurrence of \otimes_j in t different from t^ω . Suppose that $\text{ip}(t)$ is a traversal of $\tau(M)$ and t^ω appears in $t \upharpoonright^+ r_j$.*

If $t \upharpoonright^+ r_j$ is a traversal of $\tau(M^{(\otimes_j)})$ and its last node is visited using a rule different from (InputVar) and (InputVar^{val}) then t is a traversal of $\tau(M)$.

Proof. Let $t_j = t \upharpoonright^+ r_j$. Since $\text{ip}(t)$ is a traversal of M , by Prop. 4.1.5 $\text{ip}(t_j) = \text{ip}(t) \upharpoonright^+ r_j$ is a traversal of $\tau(M^{(\otimes_j)})$. We proceed by case analysis on the last rule used to produce the traversal t_j and we show that t is a traversal of M :

- (Empty), (Root). These cases do not occur since $|t_j| \geq 2$. Indeed, t_j contains at least t^ω and r_j which are two different occurrences.

- (Lam) We have $t_j = \dots \cdot \lambda \bar{\xi} \cdot n$.

Since $t_j \sqsubseteq t$, the node $\lambda \bar{\xi}$ also occurs in t . Therefore using the rule (Lam) in M we can form the traversal $t_{\leq \lambda \bar{\xi}} \cdot n$. But then we have $(t_{\leq \lambda \bar{\xi}} \cdot n) \upharpoonright^+ r_j = t_{\leq \lambda \bar{\xi}} \upharpoonright^+ r_j \cdot n = t_j \leq \lambda \bar{\xi} \cdot n = t_j = t \upharpoonright^+ r_j$. Thus, since t 's last node and n both appear in $t \upharpoonright^+ r_j$, this implies that $t_{\leq \lambda \bar{\xi}} \cdot n = t$. Hence t is a traversal of M .

- (App) $t_j = \dots \cdot \lambda \bar{\xi} \cdot @ \cdot n$. The same reasoning as in the previous case permits us to conclude.

- (Value $^{\text{at} \rightarrow \lambda}$) $t_j = \dots \cdot \lambda \bar{\xi} \cdot @ \cdot v_{@} \cdot v_{\lambda \bar{\xi}}$.

Since $t_j \sqsubseteq t$, the nodes $\lambda \bar{\xi}$, $@$, $v_{@}$ and $v_{\lambda \bar{\xi}}$ all appear in t . Moreover, since $\lambda \bar{\xi}$ is lambda node appearing in $t \upharpoonright^+ r_j$, its immediate successor must also appear in $t \upharpoonright^+ r_j$. Thus the two nodes $\lambda \bar{\xi}$ and $@$ are also consecutive in t . Hence we can use the rule (Value $^{\text{at} \rightarrow \lambda}$) in the computation tree $\tau(M)$ to produce the traversal $t_{\leq v_{\lambda \bar{\xi}}} \cdot n$ and by the same reasoning as in the previous case, we conclude that necessarily $t = t_{\leq v_{\lambda \bar{\xi}}} \cdot n$.

- (Value $^{\text{var} \rightarrow \lambda}$) $t_j = \dots \cdot \lambda \bar{\xi} \cdot x \cdot v_x \cdot v_{\lambda \bar{\xi}}$. This case is identical to the previous case.

- (Value $^{\lambda \rightarrow @}$) $t_j = \dots \cdot @ \cdot \lambda \bar{z} \cdot v_{\lambda \bar{z}} \cdot v_{@}$. Same as in the previous case by observing that $@$ and $\lambda \bar{z}$ are necessarily consecutive in t .

- (InputValue) and (InputVar). By assumption these cases do not happen.

- (Var) $t_j = \dots \cdot p \cdot \lambda \bar{x} \cdot x_i \cdot \lambda \bar{\eta}_i$ for some variable $x_i \in N_{\text{var}}^{\text{at} \rightarrow}$.

In general, two nodes p and $\lambda \bar{x}$ appearing consecutively in t_j are not necessarily consecutive in t . For in M , t can “jump” from p to a node that do not belong to the subterm $M^{(\otimes_j)}$, and thus not appearing in $t_j = t \upharpoonright^+ r_j$. This situation cannot happen here, however. Indeed, suppose that $t_{\leq p}$ extends to $t_{\leq p} \cdot m$ in $\tau(M)$. All the nodes in the thread of $\lambda \bar{\eta}_i$, in t_j , are hereditary by the same initial $@$ -node α which necessarily occurs after r_j (the first node of t_j). Consequently p belongs to $N_{\text{var}}^{\text{at} \rightarrow}$ and therefore the traversal $t_{\leq p} \cdot m$ must have been formed using the rule (Var) in $\tau(M)$. Since p appears in $t \upharpoonright^+ r_j$, by Lemma 4.1.14(i), all the nodes in the thread of p in t appear in $t \upharpoonright^+ r_j$. Thus m appears in $t \upharpoonright^+ r_j$ (since by O-visibility it points in the thread of p). Hence $(t_{\leq p} \cdot m) \upharpoonright^+ r_0 = t_{\leq p} \upharpoonright^+ r_0 \cdot p \cdot m$ which implies that m is precisely the occurrence $\lambda \bar{x}$.

Hence the nodes p , $\lambda \bar{x}$, x_i and $\lambda \bar{\eta}_i$ all appear in t with the two nodes p and $\lambda \bar{x}$ appearing consecutively. We can therefore use the rule (Var) in M to form the traversal t .

- (Value $^{\lambda \rightarrow \text{var}}$) Same proof as in the previous case.

- (Σ)/(Σ -var) Same as (App) and (Var).

- (Σ -Value) Same as (Value $^{\lambda \rightarrow \text{var}}$). □

The correspondence theorem

We now state and prove the correspondence theorem for the simply typed lambda-calculus without interpreted constants ($\Sigma = \emptyset$). This theorem establishes a correspondence between the denotation of a term in the *intentional* game model and the set of traversals of its computation tree. The result extends immediately to the simply typed lambda-calculus with *uninterpreted* constants since we can regard constants as being free variables.

Theorem 4.2.2 (The Correspondence Theorem). *For any simply typed term $\Gamma \vdash M : T$, φ_M defines a bijection from $\mathcal{Trav}(M)^*$ to $\langle\langle \Gamma \vdash M : T \rangle\rangle_s$ and ψ_M defines a bijection from $\mathcal{Trav}(M)^{\dagger\otimes}$ to $\llbracket \Gamma \vdash M : T \rrbracket$:*

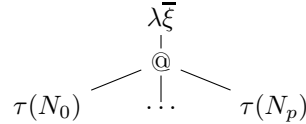
$$\begin{aligned}\varphi_M & : \mathcal{Trav}(\Gamma \vdash M : T)^* \xrightarrow{\cong} \langle\langle \Gamma \vdash M : T \rangle\rangle_s \\ \psi_M & : \mathcal{Trav}(\Gamma \vdash M : T)^{\dagger\otimes} \xrightarrow{\cong} \llbracket \Gamma \vdash M : T \rrbracket.\end{aligned}$$

REMARK 4.2.1 By corollary 4.2.1, we just need to show that φ_M and ψ_M are *surjective*, that is to say: $\varphi_M(\mathcal{Trav}(M)^*) = \langle\langle \Gamma \vdash M : T \rangle\rangle_s$ and $\psi_M(\mathcal{Trav}(M)^{\dagger\otimes}) = \llbracket \Gamma \vdash M : T \rrbracket$. Moreover the former implies the latter, indeed:

$$\begin{aligned}\llbracket \Gamma \vdash M : T \rrbracket &= \langle\langle \Gamma \vdash M : T \rangle\rangle_s \upharpoonright \llbracket \Gamma \rightarrow T \rrbracket && \text{by Eq. 4.7 from Sec. 4.2.1.5} \\ &= \varphi_M(\mathcal{Trav}(M)^*) \upharpoonright \llbracket \Gamma \rightarrow T \rrbracket && \text{by assumption} \\ &= \psi_M(\mathcal{Trav}(M)^{\dagger\otimes}) && \text{by Lemma 4.2.3(ii).}\end{aligned}$$

Therefore we just need to prove $\varphi_M(\mathcal{Trav}(M)^*) = \langle\langle \Gamma \vdash M : T \rangle\rangle_s$.

As the proof is rather technical, we begin with an overview of the argument. We proceed by induction on the structure of the computation tree. The only non-trivial case is the application; The computation tree $\tau(M)$ has the following form:



A traversal of $\tau(M)$ goes as follows: It starts at the root $\lambda\bar{\xi}$ of the tree $\tau(M)$ (rule (Root)), visits the node $@$ (rule (Lam)), and then proceeds by traversing $\tau(N_0)$ (rule (App)). While doing so, some variable y_i bound by $\tau(N_0)$'s root may be reached, in which case the traversal is interrupted by a jump to $\tau(N_i)$'s root (performed with the rule (Var)) and the process goes on with $\tau(N_i)$. Again, if the traversal encounters a variable bound by $\tau(N_i)$'s root then the traversal of $\tau(N_i)$ is interrupted and the traversal of $\tau(N_0)$ resumes. This schema is repeated until the traversal of $\tau(N_0)$ is completed⁴.

The traversal of M is therefore made of an initialization part followed by an interleaving of a traversal of N_0 and several traversals of N_i for $i = 1..p$. This schema is reminiscent of the way the evaluation copycat map ev works in game semantics.

The crucial idea of the proof is that every time the traversal jumps from one subterm to another, the jump is permitted by one of the “copycat” rules ((Var), (Value $\lambda \mapsto @$), (Value $\text{var} \mapsto \lambda$), (Value $@ \mapsto \lambda$), or (Value $\lambda \mapsto \text{var}$)). We show by a second induction that these copycat rules implement precisely the copycat evaluation strategy ev .

Proof. Let $\Gamma \vdash M : T$ be a simply typed term where $\Gamma = x_1 : X_1, \dots, x_n : X_n$. We assume that M is already in η -long normal form. By remark 4.2.1 we just need to show that $\varphi_M(\mathcal{Trav}(M)^*) = \langle\langle \Gamma \vdash M : T \rangle\rangle_s$. We proceed by induction on the structure of M :

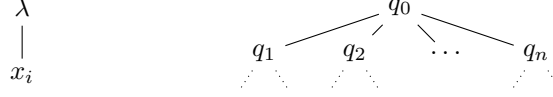
- (abstraction) $M \equiv \lambda\bar{\xi}.N : \bar{Y} \rightarrow B$ where $\bar{\xi} = \xi_1 : Y_1, \dots, \xi_n : Y_n$. On the first hand we have:

$$\begin{aligned}\langle\langle \Gamma \vdash \lambda\bar{\xi}.N : T \rangle\rangle_s &= \Lambda^n(\langle\langle \bar{\xi}, \Gamma \vdash N : B \rangle\rangle_s) \\ &\simeq \langle\langle \bar{\xi}, \Gamma \vdash N : B \rangle\rangle_s.\end{aligned}$$

On the other hand, the computation tree $\tau(N)$ is isomorphic to $\tau(\lambda\xi_1 \dots \xi_n.N)$ (up to renaming of the computation tree's root) and $\mathcal{Trav}(N)$ is isomorphic to $\mathcal{Trav}(\lambda\xi_1 \dots \xi_n.N)$. Hence we can conclude using the induction hypothesis.

⁴Since we are considering simply typed terms, the traversal does indeed terminate. However this will not be true anymore in the PCF case.

- (variable) $M \equiv x_i$. Since M is in η -long normal form, x must be of ground type. The computation tree $\tau(M)$ and the arena $\langle\langle \Gamma \rightarrow o \rangle\rangle_s$ are represented below (value leaves and answer moves are not represented):



Let π_i denote the i th projection of the interaction game semantics. We have:

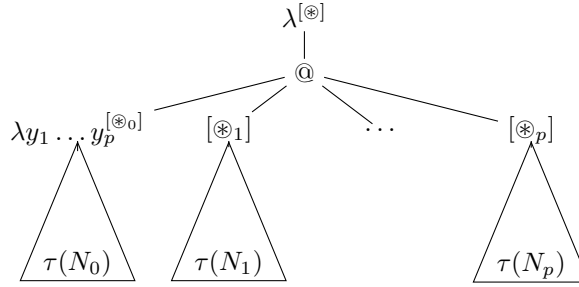
$$\langle\langle M \rangle\rangle_s = \pi_i = \text{Pref}(\{q_0 \cdot q^i \cdot v_{q^i} \cdot v_{q_0} \mid v \in \mathcal{D}\}) .$$

It is easy to see that traversals of M are precisely the prefixes of $\lambda \cdot x_i \cdot v_{x_i} \cdot v_\lambda$. M is in β -normal therefore $\text{Trav}(M)^* = \text{Trav}(M)$ and since $\varphi_M(\lambda) = q_0$ and $\varphi_M(x_i) = q^i$, we have:

$$\varphi_M(\text{Trav}(M)^*) = \varphi_M(\text{Trav}(M)) = \varphi_M(\text{Pref}(\lambda \cdot x_i \cdot v_{x_i} \cdot v_\lambda)) = \langle\langle M \rangle\rangle_s .$$

- (@-application) $M = N_0 N_1 \dots N_p : o$ where N_0 is not a variable. We have the typing judgments $\Gamma \vdash N_0 N_1 \dots N_p : o$ and $\Gamma \vdash N_i : B_i$ for $i \in 0..p$ where $B_0 = (B_1, \dots, B_p, o)$ and $p \geq 1$.

The tree $\tau(M)$ has the following form:



where \otimes_j denote the root of $\tau(N_j)$ for $j \in \{0..p\}$.

We have:

$$\langle\langle \Gamma \vdash M : o \rangle\rangle_s = \underbrace{\langle\langle \Gamma \vdash N_0 : B_0 \rangle\rangle_s, \dots, \langle\langle \Gamma \vdash N_p : B_p \rangle\rangle_s}_{\Sigma} \parallel ev$$

We refer the reader to Fig. 4.1 from the previous section for a tree-representation of $\langle\langle M \rangle\rangle_s$ which fixes the names of the different games involved in the interaction strategy. In particular the games A , B and C are defined as:

$$\begin{aligned} A &= X_1 \times \dots \times X_n \\ B &= \underbrace{((B'_1 \times \dots \times B'_p) \rightarrow o')}_{B_0} \times B_1 \times \dots \times B_p \\ C &= o . \end{aligned}$$

Let q_0 and q'_0 be the initial question of C and B_0 respectively.

\subseteq We first prove that $\langle\langle \Gamma \vdash M : T \rangle\rangle_s \subseteq \varphi_M(\text{Trav}(M)^*)$. Suppose $u \in \langle\langle \Gamma \vdash M : T \rangle\rangle_s$. We give a constructive proof that there is a traversal t such that $\varphi_M(t^*) = u$ by induction on u .

For the base case $u = \epsilon$, take t to be the empty traversal formed with (Empty). *Step case:* Suppose that $u = u' \cdot m \in \langle\langle \Gamma \vdash M : T \rangle\rangle_s$ for some move $m \in M_T$ where $u' = \varphi_M(t'^*)$ for some traversal t' of $\tau(M)$.

By unraveling the definition of $u \in \langle\langle \Gamma \vdash M : T \rangle\rangle_s$ we have:

$$\left\{ \begin{array}{l} (a) \quad u \in L_T ; \\ (b) \quad \text{For any occurrence } b \text{ in } u \text{ of an initial } B_k\text{-move, for some } k \in \{0..p\}: \\ \quad \left\{ \begin{array}{l} u \upharpoonright T^{0k} \upharpoonright b \in \langle\langle N_k \rangle\rangle_s , \\ u \upharpoonright T^{0k'} \upharpoonright b = \epsilon \quad \text{for every } k' \in \{0..p\} \setminus \{k\} ; \end{array} \right. \\ (c) \quad u \upharpoonright B_0 = u \upharpoonright B_1, \dots, B_p, C . \end{array} \right. \quad (4.8)$$

We recall that each $m \in M_T$ is an equivalence class of moves from \mathcal{M}_T . For any game A appearing in the interaction game T we will write “ $m \in A$ ” to mean that some citizen of the class m belongs to the set of moves M_A . Similarly, for any sub-interaction game T' of T , we write “ $m \in T'$ ” to mean that some citizen of the class m belongs to the set of moves $\mathcal{M}_{T'}$. We proceed by case analysis on m ; we either have $m \in C$ or $m \in T^0$. In the last case m is either in A , a superficial internal move in B or a profound internal move in B :

- Suppose $m \in C$. Moves in C are played by the standard strategy ev that does not contain any internal move. Hence m is either q_0 or v_{q_0} for some $v \in \mathcal{D}$.
 Suppose that $m = q_0$. Since q_0 can occur only once in u we have $u = q_0$ and the traversal $t = \lambda^{[\otimes]}$ formed with (Root) clearly verifies $\varphi(t^*) = u$.
 Suppose that $m = v_{q_0}$. Since m is a P-move played by the copy-cat strategy ev in B, C , it must be the copy of some move $v_{q'_0}$ that answers the question q'_0 in the sub-game o' . In fact $v_{q'_0}$ is precisely u' 's last move. Indeed suppose that $u' = \dots v_{q'_0} \cdot u''$. The play $u'_{\leq v_{q'_0}} \upharpoonright A, B$ is complete since its first move q'_0 is answered by $v_{q'_0}$. Therefore by Lemma 4.2.1(ii), $u'_{\leq v_{q'_0}} \upharpoonright T^0$ is maximal. Thus moves in u'' must be played in T^1 by ev , but since ev does not play internal moves, u'' is necessarily empty.
 Consequently, by the induction hypothesis, the last move in t' is $\varphi(v_{q'_0}) = v_{\lambda y_1}$. The rules (Value $^{\lambda \mapsto \otimes}$) and (Value $^{\otimes \mapsto \lambda}$) permits us to extend the traversal t' into $t' \cdot v_{\otimes} \cdot v_{\lambda \bar{x}}$ where v_{\otimes} and $v_{\lambda \bar{x}}$ point to the second and first node of t' respectively. Clearly we have $\varphi_M((t' \cdot v_{\otimes} \cdot v_{\lambda \bar{x}})^*) = u$.
- Suppose $m \in T^0$ and m is an initial move in B_0 . Then necessarily m is $q'_0 \in \llbracket o' \rrbracket$, the copy-cat move of the initial move $q_0 \in C$ of u . Hence $u = q_0 \cdot q'_0$. The rules (Root), (App) and (Lam) permit us to build the traversal $t = \lambda^{[\otimes]} \cdot \otimes \cdot \lambda \bar{y}^{[\otimes]}$ which clearly verifies $\varphi_M(t^*) = u$.
- Suppose $m \in T^0$ and m is an initial move in B_k for some $k \in \{1..p\}$.
 Then m is necessarily a copy-cat move played by the evaluation strategy, and the move m^1 immediately preceding m in u is an initial move of the component B'_k of B_0 .
 Thus since $\varphi_M(t'^\omega) = m^1$, t'^ω must be an occurrence of the node y_k , the k^{th} variable bound by $\lambda \bar{y}$. Hence using the rule (Var) we can form the traversal $t = t' \cdot \otimes_k$ verifying $\varphi_M(t^*) = \varphi_M(t'^*) \cdot m = u$.
- Suppose $m \in T^0$ and m is not initial in B . In $u \upharpoonright T^0$, m must be hereditarily justified by some initial move b in B_k for some $k \in \{0..p\}$. Since $u \upharpoonright T^{0k} \upharpoonright b \in \langle\langle N_k \rangle\rangle_s$, the outermost induction hypothesis gives us:

$$u \upharpoonright T^{0k} \upharpoonright b = \varphi_{N_k}(t_k^*) \quad (4.9)$$

for some traversal $t_k \in \text{Trav}(N_k)$ where w.l.o.g. we can assume that $t_k^\omega \notin V_{\otimes}$. We have:

$$\begin{aligned} \varphi_M(t_k^\omega) &= (\varphi_M(t_k^*))^\omega & (t_k^\omega \notin V_{\otimes}) \\ &= ((u' \cdot m) \upharpoonright T^{0k} \upharpoonright b)^\omega & (\text{by Eq. 4.9}) \\ &= ((u' \upharpoonright T^{0k} \upharpoonright b) \cdot m)^\omega & (m \text{ is h.j. by } b \text{ and belongs to } T^{0k}) \\ &= m . \end{aligned}$$

Take $t = t' \cdot t_k^\omega$ where t_k^ω points in t' to the image by φ_M of the occurrence justifying m in u . Since $t_k^\omega \neq @$ we have $t^* = t'^* \cdot t_k^\omega$ where t_k^ω justifier in t'^* is the same as its justifier in t .

Hence we have $\varphi_M(t^*) = \varphi_M(t'^*) \cdot \varphi_M(t_k^\omega)$ which, by the innermost I.H. together with the previous equation, is equal to $u' \cdot m$ where m 's justifier in u' corresponds to $\varphi_M(t_k^\omega)$'s justifier in $\varphi_M(t'^*)$. Consequently:

$$\varphi_M(t^*) = u \quad (4.10)$$

We are half-done at this point: it remains to show that t is indeed a traversal of $\tau(M)$. Let r_k denote the occurrence of the root \oplus_k in t that is mapped to the occurrence b in $\varphi_M(t^*)$. We make the following claim:

$$t_k = t \upharpoonright^+ r_k . \quad (4.11)$$

Indeed we have:

$$\begin{aligned} \varphi_{N_k}(t_k^*) &= u \upharpoonright T^{0k} \upharpoonright b && \text{(by Eq. 4.9)} \\ &= \varphi_M(t^*) \upharpoonright T^{0k} \upharpoonright b && \text{(by Eq. 4.10)} \\ &= \varphi_{N_k}(t^* \upharpoonright V^{(\oplus_k)} \upharpoonright r_k) && \text{(by Lemma 4.2.3(i)).} \end{aligned}$$

Since φ_{N_k} is a bijection from $\text{Trav}(N_k)^*$ to $\varphi_{N_k}(\text{Trav}(N_k)^*)$ (by Corollary 4.2.1) this implies that $t_k^* = t^* \upharpoonright V^{(\oplus_k)} \upharpoonright r_k$ which in turn is equal to $(t \upharpoonright^+ r_k)^*$ by Lemma 4.1.17 from Section 4.1.3.6. But since t_k and t do not end with an @-node, this implies equality (4.11).

We now show that t is indeed a traversal by case analysis on the rule used to visit the last occurrence of t_k in the tree $\tau(N_k)$:

- (a) Suppose the rule used to visit t_k^ω is *not* (InputVar) nor (InputVar^{val}). Then by Lemma 4.2.6, t is a traversal of M .
- (b) Suppose t_k^ω is visited with (InputVar). Then t_k is of the form

$$t_k = \dots \cdot z \cdot \dots \cdot t_k^\omega$$

for some input-variable $z \in N_{\text{var}}^{\oplus_k \vdash}$ occurring in $\downarrow t_k$ and where $t_k^\omega \in N_{\lambda}^{\oplus_k \vdash}$. Thus:

$$u = \dots \cdot \psi_{N_k}(z) \cdot \dots \cdot \psi_{N_k}(t_k^\omega) .$$

$\quad \quad \quad = m^3 \quad \quad \quad = m$

Since t_k^ω is her. enabled by the the root \oplus_k which itself is enabled by an application node, it cannot be her. enabled by the root \oplus in M . Therefore the move $\psi_{N_k}(t_k^\omega)$ is not played in A (since only node that are her. enabled by the root are mapped to move in A). Hence $\psi_{N_k}(t_k^\omega) \in B_k$. Similarly, $\psi_{N_k}(z) \in B_k$.

Now consider the top-most composition in the interaction strategy $\langle\langle M \rangle\rangle_s$: the interaction strategy $\Sigma : A \rightarrow B$ is composed with the evaluation copy-cat strategy $ev : B \rightarrow o$. Consider the sub-sequence $u \upharpoonright A, B, C$ of u consisting only of moves involved in this top-most composition (*i.e.*, the internal moves coming from other compositions at deeper level in the interaction semantics are removed). Since z is a variable node, the move $m^3 = \psi_{N_k}(z) \in B_k$ is a P-move with *respect to the game* $\llbracket A \rightarrow B_k \rrbracket$, and therefore it is an O-move in the game $\llbracket B \rightarrow o \rrbracket$. Consequently the strategy ev is responsible to play at $u_{\leq m^3} \upharpoonright A, B, C$. Let m^2 denote the move played by ev which immediately follows m^1 in $u \upharpoonright A, B, C$.

We claim that m^3 and m^2 are also consecutive in u . That is to say that no internal moves generated from the other compositions at deeper levels in the interaction

strategy can ever be played between m^3 and m^2 . Indeed, firstly the strategy ev is a pure standard strategy thus it does not play any internal move. Furthermore, suppose that the strategy Σ comes from the composition $\Sigma_l \parallel \Sigma_r$ of two interaction strategies $\Sigma_l : A \rightarrow D$ and $\Sigma_r : D \rightarrow B$ for some game D , then by the Switching Condition for function-space game [HO00] the Opponent cannot switch of component, and thus the move following m^3 in the interaction sequence $u \upharpoonright A, D, B$ must belong to B . Hence internal moves from D cannot be played immediately after m^3 .

Similarly, we can show that the move m is played by the strategy ev and is the copy of the move m^1 immediately preceding it in $u \upharpoonright A, B, C$ as well as in u .

Hence the sequence u has the following form:

$$u = \dots m^3 \cdot m^2 \cdot \dots m^1 \cdot m$$

Consequently we have:

$$t_k = \dots z \cdot \dots t_k^\omega \quad t' = \dots z \cdot \lambda \bar{y} \cdot \dots y \cdot$$

The first equation implies that t_k^ω is the i^{th} child of z in the computation tree, thus since $z \notin N^{\oplus \vdash}$, we can apply the (Var) rule to the second equation which produces the traversal of $\tau(M)$:

$$t' \cdot t_k^\omega = \dots z \cdot \lambda \bar{y} \cdot \dots y \cdot t_k^\omega$$

which is precisely the sequence t . Hence t is indeed a traversal of $\tau(M)$.

The diagram on Fig. 4.3 shows an example of such interaction sequence u .

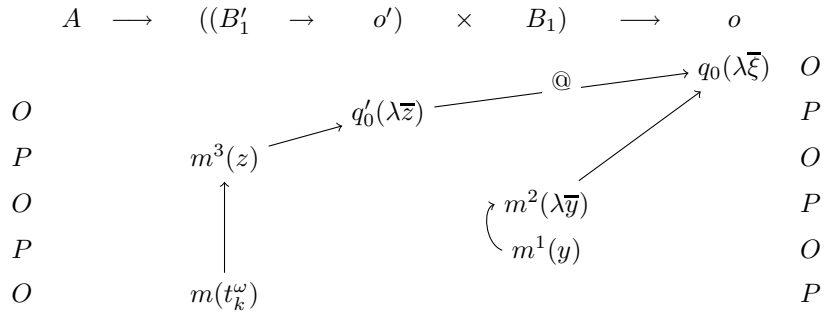


Figure 4.3: Example of a sequence $u \upharpoonright A, B, C$ for $u \in \langle\langle M \rangle\rangle_s$ and $l = 1$.

- (c) Suppose t_k 's last move is visited with the rule (**InputVar**^{val}) then the proof is the same as in the previous case but with (**InputVar**^{val}) substituted for (**InputVar**).

\supseteq The converse, $\varphi_M(\text{Trav}(M)^*) \subseteq \langle\langle M \rangle\rangle_s$, is the easy part of the proof.

Let u be as sequence of $\varphi_M(\text{Trav}(M)^*)$. Then $u = \varphi_M(t^*)$ for some traversal t of $\tau(M)$. To show that u is a position of $\langle\langle \Gamma \vdash M : T \rangle\rangle_s$ we have to prove that it satisfies the three conditions of (4.8):

- (a) $\varphi_M(t^*) \in L_T$: It suffices to show that $\varphi_M(t^*) \upharpoonright \llbracket \text{type}(T) \rrbracket$ is a legal standard position for any subtree T' of T . This reduces to showing that for any lambda node n in $\tau(M)$, $t^* \upharpoonright V^{(n)}$ is well-bracketed, satisfy visibility and alternation. But this follows immediately from the fact that t verifies these properties (since t is a traversal).
- (b) Take an initial B -move $b \in B_k$, for some $k \in \{0..p\}$, occurring in $\varphi_M(t^*)$. There is a corresponding occurrence r_k in t of a level-2 lambda node \oplus_k of $\tau(M)$. By definition, the function φ_M maps nodes from the subtree of $\tau(M)$ rooted at $\oplus_{k'}$, for

any $k' \in \{0..p\}$, to moves of the game $\langle\langle \Gamma \vdash B_{k'} \rangle\rangle_s$ that are hereditarily justified by some occurrence of $\varphi_M(\otimes_{k'})$. Hence for any $k' \in \{0..p\} \setminus \{k\}$ we clearly have $\varphi_M(t^*) \upharpoonright T^{0k'} \upharpoonright b = \epsilon$. Moreover:

$$\begin{aligned}
 u \upharpoonright T^{0k} \upharpoonright b &= \varphi_M(t^*) \upharpoonright T^{0k} \upharpoonright b \\
 &= \varphi_M(t^* \upharpoonright V^{(\otimes_k)} \upharpoonright r_k) && \text{by Lemma 4.2.3(i)} \\
 &= \varphi_M((t \upharpoonright^+ r_k)^*) && \text{by Lemma 4.1.17} \\
 &= \varphi_{N_k}((t \upharpoonright^+ r_k)^*) && \text{since } t \upharpoonright^+ r_k \text{ is a traversal of } N_k \text{ by Prop. 4.1.5} \\
 &\in \varphi_{N_k}(\mathcal{T}rav(N_k)^*) \\
 &= \langle\langle N_k \rangle\rangle_s && \text{by the induction hypothesis.}
 \end{aligned}$$

- (c) Finally we can show that $\varphi_M(t^*) \upharpoonright B_0 = \varphi_M(t^*) \upharpoonright B_1, \dots, B_p, C$ by a trivial induction on the traversal t . (This property holds because of the way the traversal rules mimic the behaviour of the evaluation strategy.)

- (Var-application) $M = x_i N_1 \dots N_p : o$.

The revealed denotation is $\langle\langle \Gamma \vdash M : o \rangle\rangle_s = \underbrace{\langle\pi_i, \langle\langle \Gamma \vdash N_1 : B_1 \rangle\rangle_s, \dots, \langle\langle \Gamma \vdash N_p : B_p \rangle\rangle_s \rangle}_{\Sigma};^{\emptyset, \{1..p\}} ev$

and the computation tree is

$$\begin{array}{c}
 \lambda^{[\otimes]} \\
 \vdots \\
 x_i \\
 \vdots \\
 \tau(N_1)^{[\otimes_1]} \quad \dots \quad \tau(N_p)^{[\otimes_p]}
 \end{array}$$

We now use the notations of Fig. 4.1 for names of the games involved in the interaction strategy. The composition of Σ with ev takes place on the following games:

$$\overbrace{X_1 \times \dots \times ((B_1'' \times \dots \times B_p'') \rightarrow o'') \dots \times X_n}^A \xrightarrow{\Sigma} \overbrace{((B_1' \times \dots \times B_p') \rightarrow o') \times B_1 \times \dots \times B_p}^B \xrightarrow{ev} \overbrace{o}^C$$

Let q_0 , q'_0 and q''_0 be the initial question of C , B_0 and X_i respectively.

$\langle\langle \Gamma \vdash M : T \rangle\rangle_s \subseteq \varphi_M(\mathcal{T}rav(M)^*)$. We show (constructively) by induction that for any $v \in \Sigma \parallel ev$, there is some traversal t such that the sequence $u = \text{cover}(v, \{0..p\}, \{0\})$ is equal to $\varphi_M(t^*)$.

The base case $v = \epsilon$ is trivial. Suppose that $v = v' \cdot m \in \Sigma \parallel ev$ where $\text{cover}(v', \{0..p\}, \{0\}) = \varphi_M(t'^*)$ for some traversal t' of $\tau(M)$ and move $m \in M_T$. Unraveling the definition of $v \in \Sigma \parallel ev$ gives

$$\left\{ \begin{array}{l} v \in L_T \\ \wedge \text{ For any occurrence } b \text{ in } v \text{ of an initial } B_k\text{-move for some } k \in \{0..p\}: \\ \quad \left\{ \begin{array}{l} v \upharpoonright T^{00} \upharpoonright b \in \pi_i \text{ if } k = 0 \text{ and } v \upharpoonright T^{0k} \upharpoonright b \in \langle\langle N_k \rangle\rangle_s \text{ if } k > 0; \\ \text{and } \forall k' \in \{0..p\} \setminus \{k\}, v \upharpoonright T^{0k'} \upharpoonright b = \epsilon \end{array} \right. \\ \wedge v \upharpoonright B_0 = v \upharpoonright B_1, \dots, B_p, C \end{array} \right. \quad (4.12)$$

We proceed by case analysis on m . It is either played in A , B or C .

1. $m \in C$. The proof is the same as in the $@$ -application case except that the rules $(\text{Value}^{\lambda \mapsto \text{var}})$ and $(\text{Value}^{\text{var} \mapsto \lambda})$ are used instead of $(\text{Value}^{\lambda \mapsto @})$ and $(\text{Value}^{@ \mapsto \lambda})$ respectively.
2. m is a superficial internal B -move. Then $\text{cover}(v, \{0..p\}, \{0\}) = \text{cover}(v', \{0..p\}, \{0\})$ so we can directly conclude from the I.H.

3. m is a profound internal B -move. Then necessarily m belongs to B_k for some $k \in \{1..p\}$ (since π_i does not contain internal moves). Thus m must be hereditarily justified by some $b \in B_k$. The treatment of this case is identical to the @-application case where $m \in T^0$ is not initial in B and $b \in B_k$ for some $k \in \{0..p\}$.
4. $m \in A$. Let b denote the initial B_k -move that hereditarily justifies m for some $k \in \{0..p\}$. If $k > 0$ then the treatment is the same as in case 3. Otherwise $b \in B_0$:
 - Suppose m is an occurrence of the initial o'' -move q_0'' . Then m is played by π_i and therefore is the copy of q_0' itself the copy of the initial move q_0 of v . Thus $v = q_0 \cdot q_0' \cdot q_0''$ and $u = q_0 \cdot q_0''$. The traversal $t = \lambda^{[\oplus]} \cdot x_i$ formed using the rules (Root) and (Lam) meets the requirement.
 - Otherwise since $v \upharpoonright b \in \pi_i$ we have $v \upharpoonright b \upharpoonright X_i = v \upharpoonright b \upharpoonright B_0$ therefore m must necessarily be hereditarily justified by the *first* occurrence of q_0'' in v .
- * Suppose m is an \bullet -question.

Then the preceding move in v is necessarily a \circ -move also played in A by the strategy π_i and therefore it is also hereditarily justified by the first occurrence of q_0'' .

Then by definition of φ_M , the last node in t' is a variable node (if the preceding move is a \circ -question) or a value-leaf of a lambda node (if the preceding move is a \circ -answer) that is hereditarily justified by the node x_i . Hence the rule (InputVar) can be applied at t' .

Let m' be m 's justifier in v' and α' be the corresponding node in t' that φ_M maps to m' . Suppose m is the i th move enabled by m' in the arena and let α be the i th child node of α' in $\tau(M)$. By definition of φ_M we have $\varphi_M(\alpha) = m$. We want to show that (InputVar) can extend t' with α . Since we have $v \upharpoonright A, C \in \llbracket M \rrbracket$, by O-visibility m' appears in $\perp v' \upharpoonright A, C \perp$, and by the induction hypothesis we have $v' \upharpoonright A, C = \psi_M(t' \upharpoonright r)$. Hence

$$\begin{aligned}
 m' \in \perp \psi_M(t' \upharpoonright r) \perp &= \psi_M(\perp t' \upharpoonright r \perp) \\
 &= \varphi_M(\perp t' \upharpoonright r \perp) \quad \text{since } \varphi_M \text{ and } \psi_M \text{ coincide on } V^{\oplus+}, \\
 &= \varphi_M(\perp t' \perp) \quad \text{by Lemma 4.1.18.}
 \end{aligned}$$

This implies that α' appears in $\perp t' \perp$ which allows us to use the rule (InputVar) to form the traversal $t = t' \cdot \alpha$ verifying $\varphi_M(t^*) = \text{cover}(v, \{0..p\}, \{0\})$.

- * Suppose m is a \circ -answer. The same argument holds using the rule (InputValue) instead of (InputVar).
- * Suppose m is an \bullet -question. We proceed identically using the rule (Lam) instead of (InputVar). The justification that α' appears in the P-view $\ulcorner t' \urcorner$ is as follows: Let $\ulcorner v \urcorner$ denote the *core* of the interaction sequence v [McC96b]. By P-visibility in $v \upharpoonright A, C$, m occurs in $\ulcorner v' \urcorner \upharpoonright A, C \urcorner$. Further we have $\ulcorner v' \urcorner \upharpoonright A, C \urcorner = \ulcorner v' \urcorner \upharpoonright A, C$ [McC96b], and clearly $\ulcorner v' \urcorner \upharpoonright A, C$ equals $\ulcorner \text{cover}(v', \{0..p\}, \{0\}) \urcorner \upharpoonright A, C$. Hence

$$m' \in \ulcorner \varphi_M(t'^*) \urcorner \upharpoonright A, C \sqsubseteq \ulcorner \varphi_M(t'^*) \urcorner.$$

This implies that α' occurs in $\ulcorner t'^* \urcorner$ which is a subsequence of $\ulcorner t' \urcorner$ by Eq. 4.1 (Sec. 4.1.3.5).

- * If m is a \circ -answer then we use the rule (Value) instead.

$\varphi_M(\text{Trav}(M)^*) \subseteq \llbracket M \rrbracket_s$. Let t be some traversal of $\tau(M)$. To show that $\varphi_M(t^*)$ is a position of $\llbracket \Gamma \vdash M : T \rrbracket_s$ we have to prove that $\varphi_M(t^*) = \text{cover}(v, \{0..p\}, \{0\})$ for some v satisfying condition (4.12).

It suffices to take $v = \delta(\varphi_M(t^*))$ where δ denotes the function sketched in Sec. 4.2.1.4 that transforms plays of the syntactically-revealed semantics to their corresponding plays of the fully-revealed semantics. The rest of the argument is the same as in the @-application case. \square

Corollary 4.2.3. *If M is in β -normal form then for any traversal t , $\varphi_M(t)$ is a maximal play if and only if t is a maximal traversal.*

Proof. If M is in β -normal form then $\text{Trav}(M)^{\dagger\otimes} = \text{Trav}(M)$ therefore φ defines a bijection on $\text{Trav}(M)$. Let t be a traversal such that $\varphi(t)$ is a maximal play. Let t' be a traversal such that $t \leq t'$. By monotonicity of φ we have $\varphi(t) \leq \varphi(t')$ which implies $\varphi(t) = \varphi(t')$ by maximality of $\varphi(t)$ which in turn implies $t' = t$ by injectivity of φ . The other direction is proved identically using injectivity and monotonicity of φ^{-1} . \square

The diagram on Fig. 4.4 recapitulates the main results of this section.

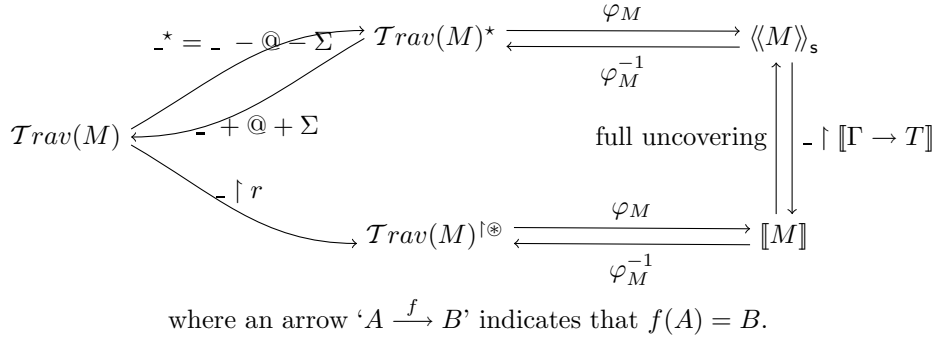
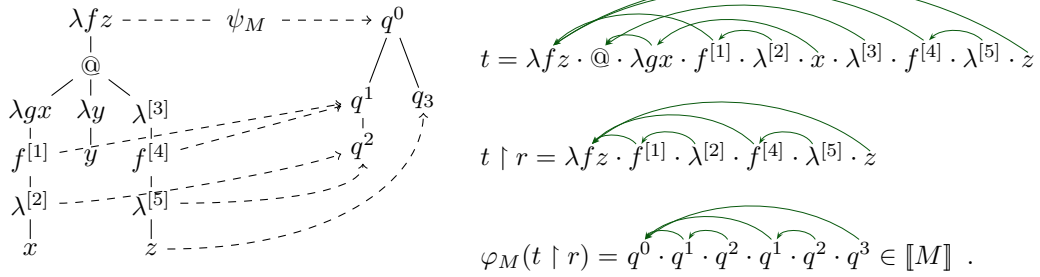


Figure 4.4: Transformations involved in the Correspondence Theorem.

Example 4.2.5. Take $M = \lambda fz.(\lambda gx.fx)(\lambda y.y)(fz) : ((o, o), o, o)$. The figure below represents the computation tree (left tree), the arena $\llbracket ((o, o), o, o) \rrbracket$ (right tree) and ψ_M (dashed line). (Only question moves are shown for clarity.) The justified sequence of nodes t defined hereunder is an example of traversal:



REMARK 4.2.2 Observe that the way we have defined traversals, the Opponent, contrary to the Proponent, is not required to play deterministically, let alone innocently. It is only required that he plays visibly (*i.e.*, his justifiers must appear in the O-view) and respects well-bracketing. This means that the game-denotation given by the Correspondence Theorem also accounts for contexts that are not simply typed terms. This indeed corresponds to the standard innocent game model of PCF: the morphisms of the category \mathcal{C}_{ib} are P-innocent strategies but not O-innocent. The addition of O-knowing-plays in the denotations is conservative for observational equivalence because the full-abstraction result holds in the category quotiented by the intrinsic preorder, and in the definition of the preorder, the “test” strategy α ranges over innocent strategies only.

4.3 Extension to PCF and IA

In this section, we show how to extend the game-semantic correspondence established for the lambda-calculus to other languages such as PCF and IA.

4.3.1 PCF fragment

The Y combinator needs a special treatment. In order to deal with it, we use an idea from Abramsky and McCusker's tutorial on game semantics [AM98b]: we consider the sublanguage PCF_1 of PCF in which the only allowed use of the Y combinator is in terms of the form $Y(\lambda x^A.x)$ for some type A . We will write Ω_A to denote the non-terminating term $Y(\lambda x^A.x)$ for a given type A .

We introduce the *syntactic approximants* to $Y_A M$:

$$\begin{aligned} Y_A^0 M &= \Gamma \vdash \Omega_A : A \\ Y_A^{n+1} M &= M(Y^n M) . \end{aligned}$$

For any PCF term M and natural number n , we define M_n to be the PCF_1 term obtained from M by replacing each subterm of the form YN with $Y^n N_n$. We have $\llbracket M \rrbracket = \bigcup_{n \in \omega} \llbracket M_n \rrbracket$ [AM98b, lemma 16].

4.3.1.1 Computation tree

In order to define the notion of computation tree for PCF terms, we first extend the inductive definition of computation tree for simply typed term (Def. 4.1.2) to PCF_1 terms by adding the new case:

$$\tau(\Omega_{(A_1, \dots, A_n, o)}) = \lambda x_1^{A_1} \dots \lambda x_n^{A_n} . \perp$$

where \perp is a special constant representing the non-terminating computation of ground type Ω_o .

We now introduce a partial order on the set of trees. A *tree* t is a labelling function $t : T \rightarrow L$ where T , called the domain of t and written $\text{dom}(t)$, is a non-empty prefix-closed subset of some free monoid X^* and L denotes the set of possible labels. Intuitively, T represents the structure of the tree (the set of all paths) and t is the labelling function mapping paths to labels. Trees are ordered using the *approximation ordering* [KNU02, section 1]: we write $t' \sqsubseteq t$ if the tree t' is obtained from t by replacing some of its subtrees by \perp . Formally:

$$t' \sqsubseteq t \iff \text{dom}(t') \subseteq \text{dom}(t) \wedge \forall w \in \text{dom}(t'). (t'(w) = t(w) \vee t'(w) = \perp) .$$

The set of all trees together with the approximation ordering form a complete partial order.

Here we take L to be the set of labels constituted of the Σ -constants, $@$, the special constant \perp , variables, and abstractions of any sequence of variables. It is easy to check that the sequence of computation trees $(\tau(M_n))_{n \in \omega}$ is a chain. We can therefore define the **computation tree** of a PCF term M to be the least upper-bound of the chain of computation trees of its approximants:

$$\tau(M) = \bigcup_{n \in \omega} (\tau(M_n))_{n \in \omega} .$$

In other words, we construct the computation tree by expanding ad infinitum any subterm of the form YM . Thus for a term of the form $Y_A F$ with $A = (A_1, \dots, A_n, o)$, the computation tree is the unique (up to alpha-conversion) infinite tree that is solution of the equation:

$$\tau(Y_A F) = \lambda \bar{x}^{\bar{A}} . \tau(F) \quad \tau(Y_A F) \quad \tau(x_1) \dots \tau(x_n) \quad (4.13)$$

where $\bar{x} = x_1 \dots x_n$ are fresh variables.

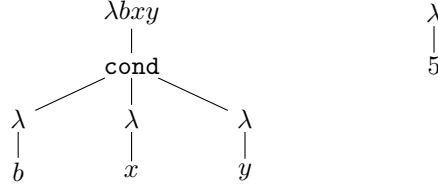
We will write (CT, \sqsubseteq) to denote the set of computation trees of PCF terms ordered by the approximation ordering \sqsubseteq defined above. Clearly (CT, \sqsubseteq) is also a complete partial order.

Example 4.3.1. Take $M = Y(\lambda f x.f x)$ where $f : (o, o)$ and $x : o$. Its computation tree $\tau(M)$, is the tree representation of the η -normal form of the infinite term $(\lambda f x.f x)((\lambda f x.f x)((\lambda f x.f x)(\dots$

It is the unique (up to alpha conversion) solution of the following equation on trees:

$$\tau(M) =$$

The remaining operators of PCF are treated as standard constants and the corresponding computation tree is constructed from the η -normal form of the term in the standard way. For instance the diagram below shows the computation tree for **cond** b x y (left) and $\lambda x.5$ (right):



The node labelled 5 has, like any other node, children value-leaves which are not represented on the diagram above for simplicity.

4.3.1.2 Traversal

New traversal rules are added to interpret the constants of PCF. The arithmetic constants are traversed as follows:

- (Nat) If $t \cdot n$ is a traversal where n denotes a node labelled with some numeral constant $i \in \mathbb{N}$ then $t \cdot n \cdot i_n$ is also a traversal where i_n denotes the value-leaf of n corresponding to the value $i \in \mathbb{N}$.
- (Succ) If $t \cdot \text{succ}$ is a traversal and λ denotes the only child node of **succ** then $t \cdot \text{succ} \cdot \lambda$ is also a traversal.
- (Succ') If $t_1 \cdot \text{succ} \cdot \lambda \cdot t_2 \cdot i_\lambda$ is a traversal for $i \in \mathbb{N}$ then $t_1 \cdot \text{succ} \cdot \lambda \cdot t_2 \cdot i_\lambda \cdot (i+1)_{\text{succ}}$ is also a traversal.

The rules for **pred** are defined similarly.

In the computation tree, nodes labelled with **cond** have three children nodes numbered from 1 to 3 corresponding to the three parameters of the operator **cond**. The traversal rules are:

- (Cond-If) If $t_1 \cdot \text{cond}$ is a traversal and λ denotes the first child of **cond** then $t_1 \cdot \text{cond} \cdot \lambda$ is also a traversal.
- (Cond-ThenElse) If $t_1 \cdot \text{cond} \cdot \lambda \cdot t_2 \cdot i_\lambda$ then $t_1 \cdot \text{cond} \cdot \lambda \cdot t_2 \cdot i_\lambda \cdot \lambda$ is also a traversal.
- (Cond') If $t_1 \cdot \text{cond} \cdot t_2 \cdot \lambda \cdot t_3 \cdot i_\lambda$ for $k = 2$ or $k = 3$ then $t_1 \cdot \text{cond} \cdot t_2 \cdot \lambda \cdot t_3 \cdot i_\lambda \cdot i_{\text{cond}}$ is also a traversal.

It is easy to verify that these traversal rules are all well-behaved. This completes the definition of traversals for the PCF.

4.3.1.3 Interaction semantics

We recall that the definition of the syntactically-revealed semantics (Sec. 4.2.1, Def. 4.2.7) accounts for the presence of interpreted constants: For any Σ -constant $f : (A_1, \dots, A_p, B)$ in the language, the revealed strategy of a term of the form $\lambda \bar{\xi}. f N_1 \dots N_p$ is defined as:

$$\langle\langle \lambda \bar{\xi}. f N_1 \dots N_p \rangle\rangle = \langle\langle N_1 \rangle\rangle, \dots, \langle\langle N_p \rangle\rangle \circ^{0..p-1} \llbracket f \rrbracket$$

where $\llbracket f \rrbracket$ is the standard strategy denotation of f .

4.3.1.4 Correspondence theorem

We now show how to extend the Correspondence Theorem of the simply typed lambda calculus (Theorem 4.2.2) to PCF.

Lemma 4.3.1. *Let (S, \subseteq) denote the set of sets of justified sequences of nodes ordered by subset inclusion. The function $\mathcal{T}rav(_)^\dagger : (CT, \sqsubseteq) \rightarrow (S, \subseteq)$ is continuous.*

Proof. Monotonicity: Let T and T' be two computation trees such that $T \sqsubseteq T'$ and let t be some traversal of T . Traversals ending with a node labelled \perp are maximal therefore \perp can only occur at the last position in a traversal. We prove the following properties:

- (i) If $t = t \cdot n$ with $n \neq \perp$ then t is a traversal of T' ;
- (ii) if $t = t_1 \cdot \perp$ then $t_1 \in \mathcal{T}rav(T')$.

(i) By induction on the length of t . It is trivial for the empty traversal. Suppose that $t = t_1 \cdot n$ is a traversal with $n \neq \perp$. By the induction hypothesis, t_1 is a traversal of T' .

We observe that for all traversal rules, the traversal produced is of the form $t_1 \cdot n$ where n is defined to be a child node or value-leaf of some node m occurring in t_1 . Moreover, the choice of the node n only depends on the traversal t_1 (for the constant rules, this is guaranteed by assumption (WB)).

Since $T \sqsubseteq T'$, any node m occurring in t_1 belongs to T' and the children nodes and leaves of m in T also belong to the tree T' . Hence n is also present in T' and the rule used to produce the traversal t of T can be used to produce the traversal t of T' .

(ii) \perp can only occur at the last position in a traversal therefore t_1 does not end with \perp and by (i) we have $t_1 \in \mathcal{T}rav(T')$.

Hence we have:

$$\begin{aligned} \mathcal{T}rav(T)^\dagger &= \{t \upharpoonright r \mid t \in \mathcal{T}rav(T)\} \\ &= \{(t \cdot n) \upharpoonright r \mid t \cdot n \in \mathcal{T}rav(T) \wedge n \neq \perp\} \cup \{(t \cdot \perp) \upharpoonright r \mid t \cdot \perp \in \mathcal{T}rav(T)\} \\ \text{(by (i) and (ii))} \quad &\subseteq \{(t \cdot n) \upharpoonright r \mid t \cdot n \in \mathcal{T}rav(T') \wedge n \neq \perp\} \cup \{t \upharpoonright r \mid t \in \mathcal{T}rav(T')\} \\ &= \mathcal{T}rav(T')^\dagger. \end{aligned}$$

Continuity: Let $t \in \mathcal{T}rav(\bigcup_{n \in \omega} T_n)$. We write t_i for the finite prefix of t of length i . The set of traversals is prefix-closed therefore $t_i \in \mathcal{T}rav(\bigcup_{n \in \omega} T_n)$ for any i . Since t_i has finite length we have $t_i \in \mathcal{T}rav(T_{j_i})$ for some $j_i \in \omega$. Therefore we have:

$$\begin{aligned} t \upharpoonright r &= \left(\bigvee_{i \in \omega} t_i \right) \upharpoonright r && \text{(the sequence } (t_i)_{i \in \omega} \text{ converges to } t) \\ &= \bigcup_{i \in \omega} (t_i \upharpoonright r) && (_ \upharpoonright r \text{ is continuous, Lemma 4.1.1)} \\ &\in \bigcup_{i \in \omega} \mathcal{T}rav(T_{j_i})^\dagger && (t_i \in \mathcal{T}rav(T_{j_i})) \end{aligned}$$

$$\subseteq \bigcup_{i \in \omega} \mathcal{Trav}(T_i)^{\dagger \otimes} \quad (\text{since } \{j_i \mid i \in \omega\} \subseteq \omega).$$

Hence $\mathcal{Trav}(\bigcup_{n \in \omega} T_n)^{\dagger \otimes} \subseteq \bigcup_{n \in \omega} \mathcal{Trav}(T_n)^{\dagger \otimes}$. \square

Proposition 4.3.1. *Let $\Gamma \vdash M : T$ be a PCF term and r be the root of $\tau(M)$. Then:*

- (i) $\varphi_M(\mathcal{Trav}(M)^*) = \langle\langle M \rangle\rangle$,
- (ii) $\varphi_M(\mathcal{Trav}(M)^{\dagger \otimes}) = \llbracket M \rrbracket$.

Proof. We first prove the result for PCF_1 : (i) The proof is an induction identical to the proof of Theorem 4.2.2. However we need to complete the case analysis with the Σ -constant cases:

- The cases **succ**, **pred**, **cond** and numeral constants are straightforward.
- Suppose $M = \Omega_o$ then $\mathcal{Trav}(\Omega_o) = \text{Pref}(\{\lambda \cdot \perp\})$ therefore $\mathcal{Trav}(\Omega_o)^{\dagger \otimes} = \text{Pref}(\{\lambda\})$ and $\llbracket \Omega_o \rrbracket = \text{Pref}(\{q\})$ with $\varphi(\lambda) = q$. Hence $\llbracket \Omega_o \rrbracket = \varphi(\mathcal{Trav}(\Omega_o)^{\dagger \otimes})$.

(ii) is a direct consequence of (i) and the Projection Lemma 4.2.3.

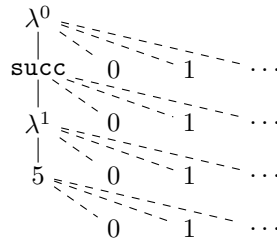
We now extend the result to PCF. Let M be a PCF term, we have:

$$\begin{aligned} \llbracket M \rrbracket &= \bigcup_{n \in \omega} \llbracket M_n \rrbracket && ([AM98b, \text{lemma 16}]) \\ &= \bigcup_{n \in \omega} \mathcal{Trav}(\tau(M_n))^{\dagger \otimes} && (M_n \text{ is a } PCF_1 \text{ term}) \\ &= \mathcal{Trav}\left(\bigcup_{n \in \omega} \tau(M_n)\right)^{\dagger \otimes} && (\text{by continuity of } \mathcal{Trav}(_)^{\dagger \otimes}, \text{ Lemma 4.3.1}) \\ &= \mathcal{Trav}(\tau(M))^{\dagger \otimes} && (\text{by definition of } \tau(M)) \\ &= \mathcal{Trav}(M)^{\dagger \otimes} && (\text{abbreviation}) \end{aligned}$$

Hence by corollary 4.2.1, φ defines a bijection from $\mathcal{Trav}(M)^{\dagger \otimes}$ to $\llbracket M \rrbracket$:

$$\varphi : \mathcal{Trav}(M)^{\dagger \otimes} \xrightarrow{\cong} \llbracket M \rrbracket.$$

Example 4.3.2 (Successor operator). Consider the term $M = \text{succ } 5$ whose computation tree is represented below. The value-leaves are also represented on the diagram, they are the vertices attached to their parent node with a dashed line.



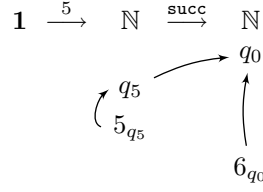
The following sequence of nodes is a traversal of $\tau(M)$:

$$t = \lambda^0 \cdot \text{succ} \cdot \lambda^1 \cdot 5 \cdot 5_5 \cdot 5_{\lambda^1} \cdot 6_{\text{succ}} \cdot 6_{\lambda^0}.$$

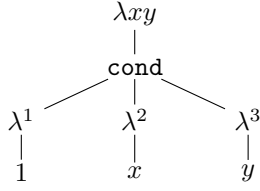
The subsequences t^* and $t \upharpoonright r$ are given by:

$$t^* = \lambda^0 \cdot \lambda^1 \cdot 5_{\lambda^1} \cdot 6_{\lambda^0} \quad \text{and} \quad t \upharpoonright r = \lambda^0 \cdot 6_{\lambda^0}.$$

We have $\varphi(t^*) = q_0 \cdot q_5 \cdot 5_{q_5} \cdot 5_{q_0}$ and $\varphi(t \upharpoonright r) = q_0 \cdot 5_{q_0}$ where q_0 and q_5 denote the roots of two flat arenas over \mathbb{N} . These two sequences of moves correspond to some play of the interaction semantics and the standard semantics respectively. The interaction play is represented below:



Example 4.3.3 (Conditional).



Take the computation tree represented on the left (value-leaves are not shown). For any value $v \in \mathcal{D}$ we have the following traversal:

$$t = \lambda xy \cdot \text{cond} \cdot \lambda^1 \cdot 1 \cdot 1_1 \cdot \lambda^3 \cdot y \cdot v_y \cdot v_{\lambda^3} \cdot v_{\text{cond}} \cdot v_{\lambda xy} \ .$$

Figure 4.5: The computation tree of the term $\lambda xy.\text{cond } 1 \ x \ y$.

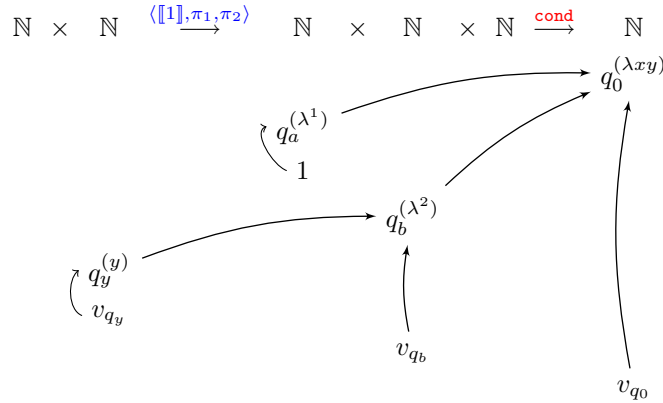
The subsequence t^* is given by:

$$t^* = \lambda xy \cdot \lambda^1 \cdot \lambda^3 \cdot y \cdot v_y \cdot v_{\lambda^3} \cdot v_{\lambda xy}$$

and the reduction $t \upharpoonright \otimes$ is given by:

$$t \upharpoonright \otimes = \lambda xy \cdot y \cdot v_y \cdot v_{\lambda xy} \ .$$

The correspondence theorem tells us that the sequence of moves $\varphi(t^*)$ (represented in the diagram below) is a play of the revealed semantics, and the sequence $\varphi(t \upharpoonright \otimes)$ is a play of the standard semantics that is obtained by hiding the internal moves from $\varphi(t^*)$.



REMARK 4.3.1 (Finite representation of the computation tree) Due to the presence of the Y combinator, computation trees of PCF terms are potentially infinite. Instead it is possible to give an equivalent finite representation of the syntax of the term using computation *graphs*. We briefly describe here how can this be achieved.

The idea is to replace Y-recursion by μ -recursion: each subterm of the form $Y_A M$ is replaced by $\mu f.Mf$ for f fresh. The computation graphs is then obtained from the eta-long normal form of the term. The abstraction nodes are generalized to take into account μ binders: an abstraction node is of the form $\lambda \bar{x}$ where \bar{x} is a list of μ -bound and λ -bound variables where the μ -bound variables are in parenthesis to distinguish them from λ -bound variables.

The computation graph of $Y_A(\lambda f^A.M)$ for $A = (A_1, \dots, A_n, o)$ is then obtained from the syntax representation of $\lambda(f)x_1 \dots x_n.[M]$ by adding a child edge going from each occurrence of the recursion variable f in $[M]$ to the root $\lambda(f)x_1 \dots x_n$.

This presentation also accounts for ground type recursion, for instance the computation graph of the **while** operator of Idealized Algol defined as $\text{while } C \text{ do } I \equiv Y(\lambda f. \text{cond } C \text{ skip } (\text{seq } I f))$ is given by the graph of $\lambda(f). \text{cond } C \text{ skip } (\text{seq } I f)$.

The order of a generalized abstraction node is still defined as the order of the term represented by the subtree rooted at this node. In other word, the order of $\lambda\bar{x}$ is defined as the order of $\lambda\bar{y}$ where \bar{y} is the sublist of \bar{x} obtained by removing all the recursion variables (those in parenthesis).

Bound variables in a generalized abstraction node $\lambda\bar{x}$ are numbered as follows: the i^{th} λ -bound variable in \bar{x} is denoted by i and the i^{th} recursion variable is denoted by (i) . The links in a justified sequence of nodes are labelled accordingly.

The notion of traversal can be modified accordingly: All the traversal rules are kept unmodified. The recursion variables in the λ -nodes are ignored by the rules since such variables are numbered differently from standard variables. In particular, the (Var) rules only applies to non-recursion variables. We only need to add a rule to handle recursion variable: whenever a traversal meets a recursion variable f in the subgraph $\tau(F)$ then instead of acting as if it was a free variable, the traversal jumps to the root of the graph:

(Var_{rec}) If $t' \cdot n \cdot \lambda\bar{x} \dots f_i$ is a traversal for some *recursion* variable f_i bound by $\lambda\bar{x}$ then so is $t' \cdot n \cdot \lambda\bar{x} \dots f_i \cdot \lambda\bar{x}$.

The enabling relation \vdash needs to be adapted to allow the root to be justified by a recursion variable (as if it was a child of the recursion variable). Also, it is now possible for a traversal to visit the root multiple times, so the definition of traversal reduction also needs to be slightly adapted: instead of keeping all the nodes hereditarily enabled by the root, it keeps the nodes that are hereditarily justified by an occurrence of the root with no justifier. The definition of the mapping ψ from nodes to moves remains consistent with this notion of computation tree, and it can be shown that the game-semantic correspondence still holds.

4.3.2 Idealized algol

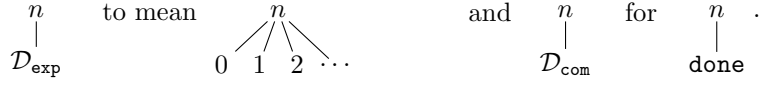
We now consider the language Idealized Algol. The general idea is the same as for PCF, however there are some difficulties caused by the presence of the two new base types **var** and **com**. We just give indications on how to adapt our framework to the particular case of IA without giving the complete proofs. However we believe that enough indications are given to convince the reader that the argument used in the PCF case can be easily adapted to IA.

Computation hypertree

The languages that we have considered up to now (lambda calculus and PCF) do not have product types. Consequently, the arenas involved in their game model only have a single initial move at most, and therefore they can be regarded as trees. This permitted us to represent the game denotation of term directly on some representation of its abstract syntax tree. In IA, however, the base type **var** is given by the product $\text{com}^{\mathbb{N}} \times \text{exp}$, and the corresponding game has an infinite number of initial moves, whereas the AST of the term is a tree and therefore has a single root.

To overcome this mismatch, we use hypertrees instead of trees to represent the syntax of the term. These hypertrees provide an abstract representation of the syntax of the term in which some nodes, called *generalized lambda nodes*, are themselves constituted of (possibly infinitely many) subnodes. Furthermore each individual subnode can have its own children nodes.

NOTATIONS 4.3.1 For any type μ , we write \mathcal{D}_μ to denote the set of values of type μ . We have $\mathcal{D}_{\text{exp}} = \mathbb{N}$, $\mathcal{D}_{\text{com}} = \{\text{done}\}$ and $\mathcal{D}_{\text{var}} = \mathcal{D}_{\text{exp}} \cup \mathcal{D}_{\text{com}}$. For any node n , if $\kappa(n)$ is of type (A_1, \dots, A_n, B) , we call B the *return type* of n . The set of value-leaves of a node n is given by \mathcal{D}_μ where μ is the return type of n . For conciseness, when representing value-leaves in the hypertree, we merge all the value-leaves of a given node of type μ into a single leaf labelled \mathcal{D}_μ . For instance the tree



The computation hypertree of a term with return type **var** has infinitely many root lambda-nodes which are merged all-together into a single node called a **generalized lambda-node**. The subnodes of a generalized lambda nodes are labelled $\lambda^r, \lambda^{w_0}, \lambda^{w_1}, \lambda^{w_2}, \dots$. Suppose that M is a term of type **var**, then the computation hypertree for $\lambda\bar{\xi}.M$ is obtained by relabelling the root λ -nodes $\lambda^r, \lambda^{w_0}, \lambda^{w_1}, \lambda^{w_2}, \dots$ into $\lambda^r\bar{\xi}, \lambda^{w_0}\bar{\xi}, \lambda^{w_1}\bar{\xi}, \lambda^{w_2}\bar{\xi}, \dots$. For a term M of type **exp** or **com**, the computation hypertree for $\lambda\bar{\xi}.M$ is computed the same way as for computation trees of lambda-terms.

Table 4.4 defines the computation hypertree for each term-construct of IA. A generalized lambda node is represented by a frame surrounding its subnodes (second and 6th row in the table).

Enabling relation, justified sequence

The notion of binder is redefined as follows: given a variable node x , the binder of x is the first node occurring in the path to the root that is a lambda node $\lambda\bar{x}$ with $x \in \bar{x}$ or a block-declaration node **new** x .

The enabling relation and the definition of justified sequence is modified so that occurrences of block-allocated variable are justified by nodes of type **new** x instead of a lambda node.

Children numbering convention

Let p be a node and suppose that its i th child n has the return type **var**. Then n is a generalized lambda-node with subnodes $\lambda^r\bar{\xi}, \lambda^{w_0}\bar{\xi}, \dots$. From the point of view of the parent node p , these subnodes are referenced as “ $i.\alpha$ ” where $0 \leq i \leq \text{arity}(p)$ and $\alpha \in \{r\} \cup \{w_k \mid k \in \mathbb{N}\}$. For instance $i.r$ refers to the root labelled $\lambda^r\bar{\xi}$ of the i th child of p , and $i.w_k$ refers to the root labelled $\lambda^{w_k}\bar{\xi}$.

Traversals

The following new rules are added on top of those defined in Sec. 4.1.3:

- *Application rules*

The rule (**app**) is now split up in three rules (**app_{exp}**), (**app_{com}**) and (**app_{var}**) corresponding to traversals ending with an @-node of return type **exp**, **com** and **var** respectively. The rules (**app_{exp}**), (**app_{com}**) are defined identically to (**app**) (See Sec. 4.1.3). The rule (**app_{var}**) is

$$(\text{app}_{\text{var}}) \quad t \cdot \lambda^k \bar{\xi} \cdot @ \in \mathcal{T}rav \text{ and } k \in \{r, w_0, w_1, \dots\} \implies t \cdot \lambda^k \bar{\xi} \cdot @ \cdot \lambda^k \bar{\eta} \in \mathcal{T}rav .$$

- *Input-variable rules*

We define the rules (**InputVal^{\$}**) for $\$$ ranging in $\{\text{com}, \text{var}, \text{exp}\}$. For **com** and **exp**, the rules are defined identically to (**InputVal**) of Sec. 4.1.3. The **var** case is implemented by two rules:

$$(\text{InputValue}_{\text{r}}^{\text{var}}) \quad \frac{t_1 \cdot \lambda^r \bar{\xi} \cdot x \cdot t_2 \in \mathcal{T}rav}{t_1 \cdot x \cdot t_2 \cdot v_x \in \mathcal{T}rav} \quad x \text{ pending node} \wedge x \in N_{\text{var}}^{\oplus} \wedge x : \text{var}, v \in \mathcal{D} .$$

$$(\text{InputValue}_{\text{w}}^{\text{var}}) \quad \frac{t_1 \cdot \lambda^w \bar{\xi} \cdot x \cdot t_2 \in \mathcal{T}rav}{t_1 \cdot x \cdot t_2 \cdot \text{done}_x \in \mathcal{T}rav} \quad x \text{ pending node} \wedge x \in N_{\text{var}}^{\oplus} \wedge x : \text{var} .$$

M	$\tau(M)$
$x : \mu$ $\mu \in \{\text{com}, \text{exp}\}$	
$\text{new } x \text{ in } N : \mu$ $\mu \in \{\text{com}, \text{exp}\}$	
$x : \text{var}$	
$\text{skip} : \text{com}$	
$\text{deref } L : \text{exp}$	
$\text{assign } L \ N : \text{com}$	
$\text{seq}_\mu \ N_1 \ N_2 : \text{com}$ $\mu \in \{\text{exp}, \text{com}\}$	
$\text{mkvar } N_w \ N_r : \text{var}$	

Table 4.4: Computation hyper-trees of IA's constructs.

$$\begin{array}{c}
\text{(deref)} \frac{t \cdot \text{deref} \in \mathcal{T}rav}{t \cdot \text{deref} \cdot n \in \mathcal{T}rav} \quad \text{(deref')} \frac{t \cdot \text{deref} \cdot n \cdot t_2 \cdot v_n \in \mathcal{T}rav}{t \cdot \text{deref} \cdot n \cdot t_2 \cdot v_n \cdot v_{\text{deref}} \in \mathcal{T}rav} \\
\text{(assign)} \frac{t \cdot \text{assign} \in \mathcal{T}rav}{t \cdot \text{assign} \cdot \lambda \in \mathcal{T}rav} \quad \text{(assign')} \frac{t \cdot \text{assign} \cdot \lambda \cdot t_2 \cdot v_\lambda \in \mathcal{T}rav}{t \cdot \text{assign} \cdot \lambda \cdot t_2 \cdot v_\lambda \cdot \lambda\bar{\eta} \in \mathcal{T}rav} \\
\text{(assign'')} \frac{t \cdot \text{assign} \cdot t_2 \cdot \lambda\bar{\eta} \cdot t_3 \cdot \text{done}_{\lambda\bar{\eta}} \in \mathcal{T}rav}{t \cdot \text{assign} \cdot t_2 \cdot \lambda\bar{\eta} \cdot t_3 \cdot \text{done}_{\lambda\bar{\eta}} \cdot \text{done}_{\text{assign}} \in \mathcal{T}rav} \\
\text{(seq)} \frac{t \cdot \text{seq} \in \mathcal{T}rav}{t \cdot \text{seq} \cdot n \in \mathcal{T}rav} \quad \text{(seq')} \frac{t \cdot \text{seq} \cdot n \cdot t_2 \cdot v_n \in \mathcal{T}rav}{t \cdot \text{seq} \cdot n \cdot t_2 \cdot v_n \cdot m \in \mathcal{T}rav} \\
\text{(seq'')} \frac{t \cdot \text{seq} \cdot t_2 \cdot m \cdot t_3 \cdot v_m \in \mathcal{T}rav}{t \cdot \text{seq} \cdot t_2 \cdot m \cdot t_3 \cdot v_m \cdot v_{\text{seq}} \in \mathcal{T}rav} \\
\text{(mkvar}_r\text{)} \frac{t \cdot \lambda^r \bar{\xi} \cdot \text{mkvar} \in \mathcal{T}rav}{t \cdot \lambda^r \bar{\xi} \cdot \text{mkvar} \cdot \lambda \in \mathcal{T}rav} \quad \text{(mkvar}'_r\text{)} \frac{t \cdot \text{mkvar} \cdot \lambda \cdot t_2 \cdot v_\lambda \in \mathcal{T}rav}{t \cdot \text{mkvar} \cdot \lambda \cdot t_2 \cdot v_\lambda \cdot v_{\text{mkvar}} \in \mathcal{T}rav} \\
\text{(mkvar}_w\text{)} \frac{t \cdot \lambda^{w_k} \bar{\xi} \cdot \text{mkvar} \in \mathcal{T}rav}{t \cdot \lambda^{w_k} \bar{\xi} \cdot \text{mkvar} \cdot \lambda\bar{\eta} \in \mathcal{T}rav} \\
\text{(mkvar}''_w\text{)} \frac{t \cdot \lambda^{w_k} \bar{\xi} \cdot \text{mkvar} \cdot \lambda\bar{\eta} \cdot t_2 \cdot \text{done}_{\lambda\bar{\eta}} \in \mathcal{T}rav}{t \cdot \lambda^{w_k} \bar{\xi} \cdot \text{mkvar} \cdot \lambda\bar{\eta} \cdot t_2 \cdot \text{done}_{\lambda\bar{\eta}} \cdot \text{done}_{\text{mkvar}} \in \mathcal{T}rav}
\end{array}$$

where v denotes some value from \mathcal{D} .

Table 4.5: Traversal rules for IA constants.

- *IA constants rules*

The rules for the constants of IA are given in Table 4.5. These rules for **new** are purely structural, they are defined similarly to $(\text{app}_{\text{exp}})$, $(\text{app}_{\text{com}})$ and $(\text{app}_{\text{done}})$.

The rules from Table 4.5 do not suffice to model **mkvar** however. We need to specify what happens when reaching a variable node that is hereditarily justified by the constant **mkvar**. Take for instance the term **assign** (**mkvar** ($\lambda x.M$) N) 7. The rule (mkvar''_w) permits one to pass the node **mkvar** and to continue with the traversal of the computation tree of $\lambda x.M$, which may subsequently lead to some occurrence of x . The behaviour of the traversal at this point is specified by the traversal rules defined in the next paragraph.

- *Variable rules*

Let x be an internal variable node. Then by definition it is either hereditarily justified by an @-node or by a Σ -constant node.

- Suppose that x 's binder is a lambda-node $\lambda \bar{x}$ and $x \in N^{\text{@}\vdash}$.

This case is a generalization of the rule (Var) (Sec. 4.1.3). The only difference is that for variables of type **var**, the lambda nodes preceding x in the traversal determines the lambda-node that will be visited next:

$$(\text{Var}_{\text{var}}) \frac{t \cdot n \cdot \lambda \bar{x} \dots \lambda^\alpha x_i \cdot x_i \in \text{Trav}}{t \cdot n \cdot \lambda \bar{x} \dots \lambda^\alpha x_i \cdot x_i \cdot \lambda \bar{\eta}_i \in \text{Trav}} \quad x_i \in N_{\text{var}}^{\text{@}\vdash} \wedge \alpha \in \{r\} \cup \{w_i \mid i \in \mathbb{N}\} .$$

- Suppose that x 's binder is a lambda-node and $x \in N^{\text{N}\Sigma\vdash}$. Then x 's binder is necessarily the second child of a **mkvar**-node (since **mkvar** is the only constant of order greater than 0).

$$(\text{mkvar-Var}) \frac{t \cdot \lambda^{w_k} \bar{\xi} \cdot \text{mkvar} \cdot \lambda x \cdot t_2 \cdot x \in \text{Trav}}{t \cdot \lambda^{w_k} \bar{\xi} \cdot \text{mkvar} \cdot \lambda x \cdot t_2 \cdot x \cdot k_x \in \text{Trav}} .$$

- Suppose that x is a block-allocated variable.

Given a block-declaration **new** x , we call *assignment of x* any segment of traversal of the form $\lambda^{w_k} \bar{\xi} \cdot x$ for some $k \in \mathcal{D}_{\text{exp}}$ and occurrence x of a node bound by **new** x . We call k the *value assigned to x* .

$$(\text{new-Var}_w) \frac{t \cdot \lambda^{w_k} \bar{\xi} \cdot x \in \text{Trav}}{t \cdot \lambda^{w_k} \bar{\xi} \cdot x \cdot \text{done}_x \in \text{Trav}} \quad x \in N_{\text{var}}^{\text{new}\vdash} .$$

$$(\text{new-Var}_r) \frac{t_1 \cdot \text{new } x \cdot t_2 \cdot \lambda^r \bar{\xi} \cdot x \in \text{Trav}}{t_1 \cdot \text{new } x \cdot t_2 \cdot \lambda^r \bar{\xi} \cdot x \cdot k_x \in \text{Trav}} \quad \text{where } k \in \mathbb{N} \text{ is the last value assigned to } x \text{ in } t_2, \text{ or } 0 \text{ if there is no such assignment.}$$

4.3.2.1 Game semantics correspondence

The properties that we proved for computation trees and traversals of the lambda-calculus with constants can easily be lifted to computation hypertrees of IA. In particular:

- Constant traversal rules are well-behaved (for order-0 and order-1 constants, this is a consequence of Lemma 4.1.3; for **mkvar** and **new** this can be easily verified);
- P-view of traversals are paths in the computation hypertrees;
- For beta-normal terms, the P-view of the reduction of a traversal is the reduction of the P-view (Lemma 4.1.20, and the O-view of a traversal is the O-view of its reduction (Lemma 4.1.18);

- There is a mapping from vertices of the computation hypertrees to moves in the interaction game semantics;
- There is a correspondence between traversals of the computation tree and plays in interaction game semantics;
- consequently, there is a correspondence between the standard game semantics and the set of justified sequences of nodes $Trav(M)^{\dagger\otimes}$.

4.4 Conclusion and related works

We have given a new presentation of game semantics based on the theory of traversals. This presentation is concrete in the sense that the traversal denotation carries syntactic information about the term. We established the connection with the Hyland-Ong game semantics by means of a Correspondence Theorem: the set of traversals of a term is isomorphic to the revealed game denotation of the term.

One advantage of the traversal theory lies in its ability to compute beta-reduction locally without having to perform term substitution. As observed by Danos et al. [DHR96], “the interaction processes at work in game semantics are implementations of *linear head reduction*”. In that regards, the traversals theory can be viewed as a rule-based implementation of the *head linear reduction strategy* [DR]. Although the idea of evaluating a term using this strategy is not new, our presentation has several advantages and novelties. Firstly, the Correspondence theorem establishes a clear correspondence with game semantics, namely that traversals gives you a way to compute precisely the revealed game denotation of a term. To our knowledge, although the notion of revealed game semantics was mentioned in previous works [Gre04], it was never formally defined. Secondly, our presentation highlights more clearly the algorithmic aspect of game semantics. The rule-based definition of traversals lends itself well to automaton characterization. An example is the characterization of higher-order recursion schemes by *collapsible higher-order pushdown automata* [HMOS08].

Another advantage of the traversal theory is its efficiency for effectively computing the game semantic denotations of a term. The traditional approach is to proceed bottom-up by appealing to compositionality. Although the compositional nature of game semantics is very attractive from a theoretical point of view, in practice it is not efficient to compute a denotation in that way. Indeed, for every subterm one has to compute all the possible ways to interact with the environment for that subterm. But this denotation is then immediately composed with another subterm, which determines part of the environment’s behaviour, thus it was wasteful in the first place to consider all the possible environment’s behaviour for the first term.

The traversal theory follows a top-down approach which means that we only consider possible behaviour of the outermost environment. Moreover contrary to the compositional method, there is no need to implement any composition mechanism: the set of traversals is just obtained by following the traversal rules; hiding of internal nodes is postponed until the end.

The lazy nature of the traversal evaluation provides a further source of efficiency: the beta-redexes are computed “on-demand” instead of performing a global substitution.

Last but not least, we believe that the syntactic correspondence between game semantics and its syntax is of pedagogical interest. Game semantics is often found hard to understand due to some obscure technical definitions. A concrete presentation such as the one given by the traversal theory, allows one to explain game-semantic concepts (such as P-view, innocence, visibility) from a programmer point of view. I have implemented a prototype tool using the F# programming language, which among other things, illustrates the theory of traversals [Blu08]. The tool lets the user “play” the game induced by a simply typed term (or a higher-order grammar) by selecting nodes from the computation tree. As the game unfolds the corresponding traversal is shown. A calculator mode allows the user to perform various operations on justified sequences. (Examples from the current chapter were all generated by this tool.)

Further correspondences

The traversal theory that we have presented here captures the lambda calculus fragment of the game model of call-by-name programming languages such as PCF and Idealized Algol. A natural way to extend this work would be to define the appropriate notion of traversal corresponding to the call-by-value games [Plo75, AM98a].

Applications

The theory of traversal has applications in several domains of research:

Verification

As mentioned previously, the theory of traversal was originally introduced by Ong to study the decidability of MSO theories of infinite trees generated by higher-order recursion scheme. More recently, Ong announced a solution to the reachability problem for PCF (Given a term M and subterm N , is there a context $C[-]$ such that evaluating $C[M]$ requires the evaluation of N ?) also based on the traversal theory.

Automata theory

The traversal theory has led to an equi-expressivity result between a certain type of automaton device called *collapsible pushdown automaton* (CPDA) and higher-order recursion schemes (HORS) [HMOS08]. One direction of this proof relies on the traversal theory: for a given HORS, a CPDA is constructed that computes precisely the set of traversals over the computation tree of the HORS.

One crucial point enabling this encoding is that structures generated by recursion scheme are of ground type. Because such structures do not interact with the environment, their game-semantic denotation is relatively simple. In particular, the O-view of the traversal does not play any role in the traversal rules and therefore the automaton does not need to calculate or remember it. A natural extension would be a similar automata-characterization for *higher-order* structures such as simply typed terms.

Pattern matching

Higher-order matching is the following problem: given an equation $M = N$ where M is an open simply typed term and N is a closed simply typed term, is there a solution substitution θ such that $M\theta$ and N have the same $\beta\eta$ -normal form? Huet conjectured in 1976 that this problem is decidable [Hue76]. It was proved only recently by Colin Stirling that it is indeed the case [Sti06].

Stirling's argument is based on a game-theoretic argument, namely the concept of tree-checking games. As pointed out by Luke Ong, Stirling's games are closely related to the innocent game semantic framework provided by the theory of traversals. The concept of traversals is implicitly present in Stirling's proof (though the notion of justification pointers is replaced by "iteratively defined look-up tables").

Analyzing syntactic constraints

The connection between syntax and semantics provided by the traversal theory enables us to analyze the effect of a given syntactic constraint on the game model. The next chapter is an example of such an application: by making simple observations about the computation tree of safe terms, the Correspondence Theorem allows us to show that their strategy denotations are of a particular kind: their plays verify a certain property called *incremental justification*.

Chapter 5

Syntactic Analysis of the Game Denotation of Safe Terms

Our aim is to characterize safety by game semantics. This chapter assumes that the reader is familiar with the basics of game semantics introduced in Chapter 2. Recall that a *justified sequence* over an arena is an alternating sequence of O-moves and P-moves such that every move m , except the opening move, has a pointer to some earlier occurrence of the move m_0 such that m_0 enables m in the arena. A *play* is just a justified sequence that satisfies Visibility and Well-Bracketing. A basic result in game semantics is that lambda-terms are denoted by *innocent strategies*, which are strategies that depend only on the *P-view* of a play. The main result (Theorem 5.4.1) of this section is that if a lambda-term is safe, then its game semantics (is an innocent strategy that) is, what we call, *P-incrementally justified*. In such a strategy, pointers emanating from the P-moves of a play are uniquely reconstructible from the underlying sequence of moves and pointers from the O-moves therein: specifically a P-question always points to the last pending O-question (in the P-view) of a greater order.

The proof of Theorem 5.4.1 relies on the Correspondence Theorem from Chapter 4 that relates the strategy denotation of a lambda-term M to the set of *traversals* over a souped-up abstract syntax tree of the η -long form of M . In the language of game semantics, this theorem says that traversals are just (concrete representations of) the *uncovering* (in the sense of Hyland and Ong [HO00]) of plays in the strategy denotation.

Since the safety condition is a syntactic constraint, it seems difficult to give a characterization in terms of game semantics, as game models are by essence syntax-independent. This is where the Correspondence Theorem comes to the rescue and help us to reason syntactically about the game denotation of a term. This ultimately permits us to give a precise game-semantic characterization of the safety restriction.

One of the main results of this chapter (Proposition 5.4.2) states that pointers in a play of a strategy denoting a safe term can be uniquely recovered from O-questions' justification pointers and from the underlying sequence of moves. In the first section we introduce the notion of *P-incrementally justified strategies*, a particular kind of strategy in which justification pointers emanating from P-moves can be reconstructed uniquely from the underlying sequences of moves and from O-moves' pointers. We then introduce the notion of *incrementally-bound computation trees* and establish a relationship between incremental-binding and P-incremental justification (Proposition 5.3.2). Finally, we show that safe simply typed terms have incrementally-bound computation trees, consequently their game denotation is P-incrementally justified.

The third section of this chapter is concerned only with the safe lambda-calculus without interpreted constants. In the following sections we extend the result by taking into account the interpreted constants of PCF and IA. We show that safe PCF and safe IA terms are denoted by P-incrementally justified strategies.

Some of the results presented in this chapter were first published in TLCA [BO07]. They are

reproduced here with complete proofs and generalized to the languages PCF and IA.

5.1 P-incrementally justified strategies

In the game-semantic literature, some authors use the term “order of a question move” to refer to the length of the path in the arena to the initial move that hereditarily enables it. For the purpose of studying the safety restriction, however, it will be convenient instead to call it the *level* of the node, and reserve the term “order” to refer to another quantity: The *order of a move* m , written $\text{ord } m$, is defined as the length of the path from m to its furthest leaf in the arena minus 1. Thus the order of an arena can be defined in term of move-order: it is precisely the greatest order of its initial moves.

Definition 5.1.1. A strategy σ is said to be *P-incrementally justified* if for any play $s q \in \sigma$ where q is a P-question, q points to the last unanswered O-question in $\lceil s \rceil$ with order strictly greater than $\text{ord } q$.

Note that although the pointer is determined by the P-view, the choice of the move itself can be based on the whole history of the play. Thus P-incremental justification does not imply innocence.

The definition suggests an algorithm that, given a play of a P-incrementally justified denotation, uniquely recovers the pointers from the underlying sequence of moves and from the pointers associated to the O-moves therein. Hence:

Lemma 5.1.1. *In P-incrementally justified strategies, pointers emanating from P-moves are superfluous.*

Proof. Suppose σ is a P-incrementally justified strategy. We prove that pointers attached to P-moves in a play $s \in \sigma$ are uniquely recoverable by induction on the length of s . *Base case:* if $|s| \leq 1$ then there is no pointer to recover. *Step case:* suppose $sm \in \sigma$. If m is an answer move then by the well-bracketing condition m points to the last unanswered question in s . If m is a P-question then by P-incremental justification of σ , m points to the last O-move in $\lceil s \rceil$ with order strictly greater than $\text{ord } q$. Since we have access to O-moves’ pointers, we can compute the P-view $\lceil s \rceil$. Hence m ’s pointer is uniquely recoverable. \square

Example 5.1.1. Copycat strategies, such as the identity strategy id_A on game A or the evaluation map $\text{ev}_{A,B}$ of type $(A \Rightarrow B) \times A \rightarrow B$, are all P-incrementally justified.¹

5.2 Dead code elimination

We recall that the β -normal form of a term of an applied lambda calculus is the (possibly infinite) term obtained by reducing all the β -redexes. Because of the presence of interpreted constants, a β -normal form is not necessarily normal with respect to the small-step semantics. For instance in PCF, the term $\text{cond } 0 \ M \ N$ is β -normal but it reduces in one step to M .

We say that a subterm N of $M : (A_1, \dots, A_n, o)$ is *dead code* if for any context $C[-]$ such that $C[M]$ is of ground type, any reduction sequence starting from $C[M]$ does not involve a reduction of the subterm N ; formally, for any reduction sequence $C[M] \equiv T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_k$, there is no $j \in \{0..k-1\}$ such that $T_j = C'[N]$ and $T_{j+1} = C'[N']$ for some evaluation context $C'[-]$ and term N' .

Example 5.2.1. The subterm N in $\text{cond } 0 \ M \ N$ is dead-code.

The dead code elimination problem is the converse of the *reachability problem*: given a term M containing a subterm N of M , is there a context $C[-]$ such that $C[M]$ is of ground type, and a reduction sequence $C[M] \equiv T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_j \equiv E[N]$ for some evaluation context $E[-]$.

¹In such strategies, a P-move m is justified as follows: either m points to the preceding move in the P-view or the preceding move is of smaller order and m is justified by the second last O-move in the P-view.



The reachability problem is clearly not trivial. In fact for PCF it is not decidable since PCF is Turing-complete and the halting problem for PCF can be encoded into a reachability problem.²

Let M be a term in eta-long normal form. Occurrences of a variables that are in the dead code of M are called **dead occurrences**. Given a term M , we define M^* as the term obtained from the (possibly infinite) η -long normal form of M by substituting all subterms of the form $xN_1 \dots N_k$ where $x : (B_1, \dots, B_k, o)$ is a dead variable occurrence by the constant \perp of type o . This process is called **dead variable elimination**. We write $\tau(M)^*$ to denote the equivalent transformation on the computation tree of M .

Clearly we have:

$$\mathcal{T}rav(M^*) \subseteq \mathcal{T}rav(M) . \quad (5.1)$$

Reachability by traversals



A node of a computation tree is said to be **reachable** if there exists a traversal that visits it. By the Correspondence Theorem, it is easy to show that if a node is not *reachable* then the corresponding variable occurrence is a dead occurrence. In particular:

Lemma 5.2.1. *If x is a variable node in $\tau(M)^*$ then the corresponding node in $\tau(M)$ is reachable by some traversal.*

However the converse does not hold. This is because the Correspondence Theorem concerns the *intentional* innocent game model where the Opponent is not restricted to play deterministically, let alone innocently. Thus in this model, the strategy denotation accounts for contexts $C[-]$ that are not part of the language considered, whereas in the definition of dead-code elimination, the context ranges over term of the language only. Hence a variable node may be reachable by a traversal (as defined in Chapter 4) but not reachable with the operational semantics of the language.

Example 5.2.2. Take the following simply typed lambda term:

$$M \equiv \lambda\varphi^{(o,o)} x^o y^o z^o. \varphi x(\varphi yz) .$$

The node corresponding to the occurrence y is reachable by the traversal $\lambda\varphi xyz \cdot \varphi \cdot \lambda \cdot \varphi \cdot \lambda \cdot y$ but there is no simply-typed context $C[-]$ such that the evaluation of $C[M]$ leads to the evaluation of y .

The two notions of reachability can be reconciliated by adding a constraint in the rules of Table 4.3 enforcing O-innocence (*i.e.*, whenever a lambda node is visited, it is uniquely determined by the O-view of the traversal at that point).

5.3 Incremental binding

In this section, we work in the general setting of an applied simply typed lambda calculus extended with a stock of interpreted constants Σ (but without recursion), whose terms are of the form $\Gamma \vdash M : T$. We consider its safe fragment, as defined in Sec 3.5.3, whose terms are written $\Gamma \vdash M : T$.

For the rest of this section we fix a term $\Gamma \vdash M : T$ of this unspecified language. We assume that the language has a fully-abstract game-semantic model. We write $\llbracket \Gamma \vdash M : T \rrbracket$ to denote the strategy denotation in the intensional model. We further assume that there are *well-behaved* (see Def. 4.1.14) traversal rules modeling the behaviour of the constants in such a way that the game-semantic correspondence (Theorem 4.2.2) holds for that language.

²When restricted to the finitary fragment, however, reachability is decidable. Luke Ong recently announced a proof of this result based on the theory of traversals.



NOTATIONS 5.3.1 We call **path** any sequence of nodes such that for any two consecutive nodes $a \cdot b$ in the sequence, a is the parent of b . We write $[n_1, n_2]$ to denote, if it exists, the unique path going from node n_1 to node n_2 equipped with the justification pointers induced by the enabling relation \vdash (A node has a unique enabler in the path to the root thus for each occurrence in $[n_1, n_2]$ there is at most one occurrence of its enabler in $[n_1, n_2]$). We write $]n_1, n_2]$ for the sub-sequence of $[n_1, n_2]$ obtained by removing n_1 together with all the associated pointers.

The symbol \otimes denotes the root of the computation tree $\tau(M)$, $N^{\otimes\vdash}$ denotes the subset of N consisting of nodes that are hereditarily enabled by \otimes , and $N^{\Sigma\vdash}$ denotes the nodes that are hereditarily enabled by some constant in Σ .

Definition

Recall from the definition of computation trees (Chapter 4) that a variable node n labelled x is said to be *bound* by a node m if m is the closest node in the path from n to the root such that m is labelled $\lambda\xi$ with $x \in \xi$. Thus the binder node always occurs in the path from the variable node that it binds to the root. We now introduce a class of computation trees in which the binder node is uniquely determined by the nodes' orders.

Definition 5.3.1 (Incrementally-bound computation tree). Let A be a subset of nodes of the computation tree.

- (i) A variable node x of a computation tree is said to be ***A-incrementally-bound*** if its enabler is the first λ -node from A in the path to the root that has order strictly greater than $\text{ord } x$. Formally:

$$x \text{ is } A\text{-incrementally-bound} \iff \begin{cases} x \text{ is enabled by } b \in [\otimes, x] \cap A ; \\ \text{ord } b > \text{ord } x \\ \forall \lambda\text{-node } n' \in]n, x] \cap A. \text{ord } n' \leq \text{ord } x . \end{cases}$$

This definition can be split into two cases:

- (a) x is *bound* by the first λ -node from A occurring in the path to the root that has order strictly greater than $\text{ord } x$.
- (b) or x is a *free variable* and all the λ -nodes from A occurring in the path to the root except the root have order smaller or equal to $\text{ord } x$.
- (ii) A computation tree is said to be ***A-incrementally-bound***, also abbreviated A-i.b., if all the variable nodes from A are *A-incrementally-bound*.
- (iii) A node (resp. a tree) is ***incrementally-bound*** if it is $(N \setminus N^{\Sigma\vdash})$ -***incrementally-bound*** where N is the entire set of nodes of the computation tree and $N^{\Sigma\vdash}$ is the set of nodes hereditarily justified by some constant node.

Lemma 5.3.1.

- (i) For any two sets of nodes A and B verifying $A \subseteq B$, *B-incremental-binding* implies *A-incremental-binding*.
- (ii) $\tau(M)$ is *A-incrementally bound* if and only if $\tau(\text{closure}(M))$ is.

where $\text{closure}(M)$ denotes the closed term obtained by abstracting the free variables in M (see Sec. 2.1).

Proof. (i) follows immediately from the definition. (ii) This is because the computation trees $\tau(M)$ and $\tau(\text{closure}(M))$ are isomorphic and the enabling relation \vdash is defined identically on these two trees (since free variable nodes are enabled by the root). \square

Safety and incremental binding

We recall that a term is *almost safe* if it can be written $\lambda x_1 \dots x_n. N_0 \dots N_p$ for some $n, p \geq 0$ where N_i is safe for all $0 \leq i \leq p$. It is an *almost safe application* if further $n = 0$.

Proposition 5.3.1 (Safe terms have incrementally-bound computation trees). *Let $\Gamma \vdash M : T$ be a term of some applied typed lambda calculus (without recursion).*

- (i) *If M is almost safe then $\tau(M)$ is incrementally-bound ;*
- (ii) *conversely, if $\tau(M)$ is incrementally-bound then the η -long normal form of M is almost safe, and safe if further M is closed.*

Proof. (i) Suppose that M is almost safe. Computation trees are defined modulo eta-long normal expansion thus since this transformation preserves almost safety (Lemma 3.1.16) we can assume that M is in eta-long nf. By the previous lemma, to show that $\tau(M)$ is incrementally-bound we just have to show that $\tau(\text{closure}(M))$ is incrementally-bound. We now consider $\tau(\text{closure}(M))$.

In an applied safe lambda-calculus, the Γ -variables with the lowest order must be all abstracted at once when applying the abstraction rule. Since the computation tree merges consecutive abstractions into a single node, any Γ -variable x occurring free in the subtree rooted at a λ -node $\lambda \bar{\xi} \notin N^{\Sigma^+}$ different from the root must have order greater or equal to $\text{ord } \lambda \bar{\xi}$. Conversely, if a lambda node $\lambda \bar{\xi}$ binds a variable node x then its order is $1 + \max_{z \in \bar{\xi}} \text{ord } z > \text{ord } x$.

Let x be a Γ -variable node in $\tau(\text{closure}(M))$. Its enabler necessarily occurs in the path to the root, therefore, according to the previous observation, x must be bound by the first λ -node occurring in $[\otimes, x] \setminus N^{\Sigma^+}$ with order strictly greater than $\text{ord } x$. Hence τ is incrementally-bound.

(ii) We first show the result for closed term. Let $\vdash M : T$ be a closed term such that $\tau(M)$ is incrementally-bound. We assume that M is already in η -long normal form. We prove by induction that M is safe. The base case $M \equiv \lambda \bar{\xi}. \alpha$ for some variable or constant α is trivial. *Step case:* $M \equiv \lambda \bar{\xi}. N_1 \dots N_p$. Let $1 \leq i \leq p$. Each N_i can be written $\lambda \bar{\eta}_i. N'_i$ where N'_i is not an abstraction. By the induction hypothesis, $\lambda \bar{\xi}. N_i \equiv \lambda \bar{\xi} \bar{\eta}_i. N'_i$ is safe which means that the term N'_i is also safe: we have $\bar{\xi}, \bar{\eta}_i \vdash_s N'_i : A_i$ for some type A_i . Let z be a variable occurring free in N'_i . Since M is closed, z is either bound by $\lambda \bar{\eta}_i$ or $\lambda \bar{\xi}$. In the latter case, since $\tau(M)$ is i.b. we have that $\text{ord } z$ is smaller than $\text{ord } \lambda \bar{\eta}_i = \text{ord } N_i$, thus in both case we are allowed to abstract the variables $\bar{\eta}_i$ using the rule (abs), which shows that N_i is safe. Since all the N_i s are safe and the term $N_1 \dots N_p : o$ is of order 0, by the rule (app) we have that $N_1 \dots N_p$ is safe: $\bar{\xi} \vdash_s N_1 \dots N_p : o$. The rule (abs) then gives us $\vdash_s \lambda \bar{\xi}. N_1 \dots N_p$.

Now if M is open, by the preceding case we have that $\text{closure}(M)$ is safe. But by “peeling-off” abstractions from a safe term we obtain an almost safe term, thus M is almost safe. \square

Note that the hypothesis that M is closed in (ii) is necessary. Take for instance the two terms $\lambda xy. x$ and $\lambda y. x$, where $x, y : o$. Their have isomorphic incrementally-bound computation trees. But $\lambda xy. x$ is safe and $\lambda y. x$ is only almost safe.

For the second part of this proposition a slightly stronger result holds if the term is β -normal and does not contain any interpreted constant:

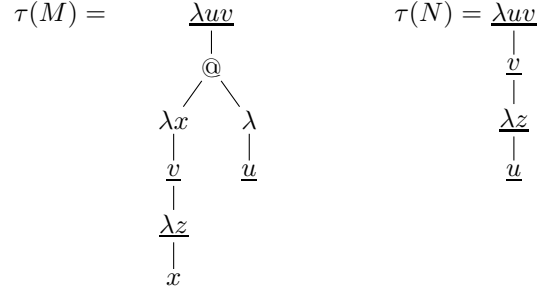
Corollary 5.3.1. *Let M be a β -normal term containing no interpreted constant. If all the input variables are incrementally-bound then the η -long normal form of M is almost safe, and safe if further M is closed.*

This is simply because in the computation tree of such terms all the variable nodes are input-variables. This stronger result does not hold for terms containing redexes: For any unsafe closed term U , the term $(\lambda u. u) U$ is unsafe but the only input-variable is u and it is incrementally-bound. It does not hold either for terms with interpreted constants: For any closed unsafe term U of type exp , the PCF term $\text{succ } U$ has no input variable but it is unsafe.

Corollary 5.3.2. *If $\tau(M)$ is incrementally-bound and $M \rightarrow_{\beta_s} N$ then $\tau(N)$ is incrementally-bound.*

Proof. Suppose that $\tau(M)$ is i.b. Then by Proposition 5.3.1(ii) the eta-long normal form of M is almost safe, therefore so is M by Lemma 3.1.16. But almost safety is preserved by β_s -reduction (Lemma 3.1.17) therefore N is almost safe, and by Proposition 5.3.1(i), $\tau(N)$ is incrementally-bound. \square

Note that this corollary cannot be generalized to A -incremental-binding for any set of node A . Take for instance the term $M \equiv \lambda u^o v^{((o,o),o)}.(\lambda x^o.v(\lambda z^o.x))u$ which beta-reduces to $N \equiv \lambda uv.v(\lambda z.u)$. The computation trees are:



If we take A to be the set of nodes that are hereditarily enabled by the root (underlined in the figure above) then $\tau(M)$ is A -incrementally-bound but $\tau(N)$ is not.

Incremental justification and incremental binding

Proposition 5.3.2 (Incremental-binding and P-incremental justification). *Let $\Gamma \vdash M : T$ be a term-in-context of some applied typed lambda calculus.*

- (i) *Suppose M is β -normal. If all the reachable input-variable nodes of the computation tree $\tau(\Gamma \vdash M : T)$ are incrementally bound then $\llbracket \Gamma \vdash M : T \rrbracket$ is P-incrementally justified.*
- (ii) *If $\llbracket \Gamma \vdash M : T \rrbracket$ is P-incrementally justified then all the reachable input-variable nodes of the computation tree $\tau(\Gamma \vdash M : T)$ are $N^{\otimes \vdash}$ -incrementally bound.*

Proof. (i) Suppose M is a β -nf. W.l.o.g we can assume that M is a closed term since the incremental-binding property is conserved when taking the closure of a term and since the denotation of the closure is isomorphic to the denotation of the term.

Suppose that all the reachable input-variable nodes of $\tau(M)$ are incrementally bound. We want to show that $\llbracket M \rrbracket$ is P-incrementally justified. Take a play $s \in \llbracket M \rrbracket$ ending with a question P-move q . By the Correspondence Theorem 4.2.2, there is a traversal t of $\tau(M)$ starting with an occurrence r of the root \otimes such that $\psi_M(t \upharpoonright r) = s$. We assume t to be the shortest such traversal, so that the last occurrence of t —name it n —is hereditarily justified by r , and is by definition an occurrence of a reachable node. Since ψ_M maps n to the P-question q , n is necessarily an occurrence of a variable node x . By Lemma 4.2.2 (iv), the P-views of s and $t \upharpoonright r$ are computed identically and have the same underlying sequence of justification pointers so in particular the node n and the move q both point to the same position in the justified sequence $\ulcorner t \upharpoonright r \urcorner$ and $\ulcorner s \urcorner$ respectively. Further by Lemma 4.2.2(iii), ψ_M maps nodes of a given order to moves of the same order. Hence showing that s is P-incrementally justified amounts to showing that n 's justifier in t is the latest lambda-node in $\ulcorner t \upharpoonright r \urcorner$ with order strictly greater than $\text{ord } n$.

Let m denote n 's justifier in t . The term M is closed therefore x is necessarily a bound variable and n is an occurrence of x 's binder in $\tau(M)$. The traversal t is incrementally-bound by assumption and n belongs to $N \setminus N^{\Sigma \vdash} = N^{\otimes \vdash}$ therefore by definition of incremental binding the occurrence m is the last λ -node in $[\otimes, n] \cap N^{\otimes \vdash}$ with order strictly greater than $\text{ord } n$. The Path-P-view correspondence (Prop. 4.1.1) gives $[\otimes, n] \cap N^{\otimes \vdash} = \ulcorner t \urcorner \upharpoonright r$ which in turn equals $\ulcorner t \upharpoonright r \urcorner$ by Lemma 4.1.20 (it is applicable because M is a β -nf and we have assumed that the constant traversals are well-behaved).

(ii) Suppose $\llbracket M \rrbracket$ is P-incrementally justified. Let x be a reachable input-variable node of $\tau(M)$: there exists a traversal of the form $t \cdot x$ in $\text{Trav}(M)$ such that x is hereditarily justified in t by the first occurrence r of $\tau(M)$'s root.

The correspondence theorem shows that $\varphi((t \cdot x) \upharpoonright r) = \varphi(t \upharpoonright r) \cdot \varphi(x)$ belongs to $\llbracket M \rrbracket$. Since $\llbracket M \rrbracket$ is P-incrementally justified, $\varphi(x)$ points to the last O-move in $\ulcorner \varphi(t \upharpoonright r) \urcorner$ with order strictly greater than $\text{ord } \varphi(x)$. Consequently x points to the last λ -node in $\ulcorner t \upharpoonright r \urcorner$ with order strictly greater than $\text{ord } x$.

But by Lemma 4.1.19, $\ulcorner t \upharpoonright r \urcorner$ contains $\ulcorner t^\top \upharpoonright r \urcorner$ as a subsequence, and by P-visibility m occurs in this subsequence, thus m is also the last λ -node in $\ulcorner t^\top \upharpoonright r \urcorner$ with order strictly greater than $\text{ord } x$. By the Path-P-view correspondence (Prop. 4.1.1) this means that m is the last λ -node in $[\otimes, x] \cap N^{\otimes+}$ with order strictly greater than $\text{ord } x$. Hence $\tau(M)$ is $N^{\otimes+}$ -incrementally-bound. \square

Corollary 5.3.3. *Let $\Gamma \vdash M : A$ be a term-in-context of some applied typed lambda calculus.*

- (i) *If $\tau(\Gamma \vdash M : A)$ is incrementally-bound then $\llbracket \Gamma \vdash M : A \rrbracket$ is P-incrementally justified;*
- (ii) *if M is β -normal and $\llbracket \Gamma \vdash M : A \rrbracket$ is P-incrementally justified then $\tau(\Gamma \vdash M : A)^*$ is incrementally-bound.*

Proof. (i) Let M' denote the beta-normal form of M . If $\tau(M)$ is incrementally bound then by Corollary 5.3.2 so is $\tau(M')$. So in particular all the *reachable* input-variable node of $\tau(M')$ are incrementally bound. Thus by Proposition 5.3.2(i), $\llbracket M \rrbracket = \llbracket M' \rrbracket$ is P-incrementally justified.

(ii) Suppose that $\llbracket M \rrbracket$ is P-incrementally justified. Consider $\tau(M)^*$. By definition, a tree is incrementally bound just if it is $N \setminus N^{\Sigma+}$ -incrementally bound. Since M is β -normal, variable nodes cannot be hereditarily enabled by an $@$ -node thus $N^{\otimes+} = N \setminus N^{\Sigma+}$. Thus to show that $\tau(M)^*$ is incrementally-bound we just need to show that its variables are $N^{\otimes+}$ -incrementally bound. But by definition its variable nodes are precisely those of $\tau(M)$ that are reachable. Hence we just need to show that the reachable input variables of $\tau(M)$ are $N^{\otimes+}$ -incrementally bound. This is precisely what Proposition 5.3.2(ii) says. \square

5.4 Safe lambda calculus

We now consider the special case of the pure (*i.e.*, without interpreted constants) safe lambda-calculus. For any simply-type term $\Gamma \vdash_{\text{st}} M : T$ we write $\llbracket \Gamma \vdash_{\text{st}} M : T \rrbracket$ to refer to the innocent game denotation of $\Gamma \vdash_{\text{st}} M : T$.

Lemma 5.4.1. *For simply typed lambda term in β -normal form, all the nodes of the computation tree are reachable by some traversal obtained using the rules of Table 4.3.*

Proof. Since M is in β -normal form, its computation tree has no application node and therefore all the variable nodes are hereditarily justified by the root. Hence each variable node can be reached by the traversal consisting of the path from the root to that node: the rule (Lam) and (InputVar) permit us to visit the variable nodes and lambda nodes respectively. \square

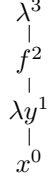
Proposition 5.4.1. *Let $\Gamma \vdash_{\text{st}} M : T$ be a pure (*i.e.*, with no interpreted constants) simply typed term in β -normal form. Then $\llbracket \Gamma \vdash_{\text{st}} M : T \rrbracket$ is P-incrementally justified if and only if the computation tree $\tau(M)$ is incrementally-bound.*

Proof. By Lemma 5.4.1, all the variable nodes are reachable in a β -normal term thus $\tau(M) = \tau(M)^*$ and the result follows from Corollary 5.3.3. \square

Example 5.4.1.

- (i) For any higher-order variable $x : A$ the computation tree $\tau(x)$ is incrementally-bound. Consequently the projection strategies are all P-incrementally justified.

- (ii) Consider the β -normal term $\Gamma \vdash_{\text{st}} f(\lambda y.x) : o$ where $y : o$ and $\Gamma = f : ((o, o), o), x : o$. The figure on the right represents its computation tree with the node orders given as superscripts. The node x is not incrementally-bound because the node x of order 0 is not bound by the order 1 node λy . Therefore $\tau(f(\lambda y.x))$ is not incrementally-bound and by Proposition 5.4.1, $\llbracket \Gamma \vdash_{\text{st}} f(\lambda y.x) : o \rrbracket$ is not P-incrementally justified. Similarly we can check that $\llbracket \lambda y.x \rrbracket$ is P-i.j. while $\llbracket f(\lambda y.x) \rrbracket$ is not.
- (iii) The previous examples show that the denotations $\llbracket \Gamma \vdash_{\text{st}} f : ((o, o), o) \rrbracket$ and $\llbracket \Gamma \vdash_{\text{st}} \lambda y.x : (o, o) \rrbracket$ are both P-i.j. whereas $\llbracket \Gamma \vdash_{\text{st}} f(\lambda y.x) : o \rrbracket$ is not. This shows that application does not preserve P-incremental justification.



The third example suggests that P-incremental justification is not a compositional property. In Chapter 6 we will identify a sufficient condition enabling compositionality of P-incrementally justified strategies.

Putting Proposition 5.4.1 and Proposition 5.3.1 together gives us a game-semantic characterization of safety. This result was first presented in TLCA2007, [BO07, Theorem 3(ii)]:

Theorem 5.4.1 (Characterization Theorem for the safe lambda calculus). *Let $\Gamma \vdash_{\text{st}} M : A$ be a pure simply typed term (with no interpreted constants).*

- (i) *If M is almost safe (and in particular if it is safe) then $\llbracket \Gamma \vdash_{\text{st}} M : A \rrbracket$ is P-incrementally justified.*
- (ii) *If $\llbracket \Gamma \vdash_{\text{st}} M : A \rrbracket$ is P-incrementally justified then the beta-normal form of M is almost safe, and safe if further M is closed.*

Proof. (i) Since M is almost safe, by Proposition 5.3.1(i), its computation tree is incrementally-bound. Hence by Corollary 5.3.3(i) its denotation is incrementally justified.

(ii) Since a term has the same denotation as its beta-normal form we can assume that M is beta-normal. By Proposition 5.4.1 its computation tree is incrementally bound, and by Proposition 5.3.1(ii), the eta-long normal form of M is safe if it is a closed term and almost safe otherwise. The same holds for M itself since both safety and unsafety are preserved by eta-long normal expansion (Lemma 3.1.16 and 3.1.2). \square

In particular, a term has a P-incrementally justified denotation if and only its beta-normal form is almost safe.

REMARK 5.4.1 In game semantics, the Opponent's strategy is dictated by the denotation of a term (the context) representing the environment so that if the language considered is a pure functional language such as PCF then the Opponent necessarily plays innocently. In the intentional game denotation, however, all possible O-moves are accounted for at every position, including those moves that would break "O-innocence". In the extensional denotation, non O-innocent plays do not have any effect since the test strategy from the intrinsic preorder ranges over P-innocent strategies.

The second part of the previous theorem crucially relies on the presence of those non O-innocent plays: it is true that an unsafe beta normal term is denoted by a non P-i.j. strategy, but the failure to satisfy P-incremental justification may only be due to some play that does not affect the extensional denotation of the term. For instance the beta-normal term $\lambda\varphi^{((o,o),o,o)} y^o. \varphi(\lambda x^o.x)(\varphi(\lambda x^o.y) y)$ is clearly unsafe and, as is implied by (ii), its denotation in the intentional game model is not P-i.j. since for instance the last node in the traversal $t = \lambda\varphi y \cdot \varphi^1 \cdot \lambda \cdot \varphi^2 \cdot \lambda x \cdot y$ is not incrementally justified. But the traversal t corresponds to a play that does not respect O-innocence since we have $\perp t_{\leq \varphi^1} \perp = \perp t_{\leq \varphi^2} \perp$ and the node visited after φ^1 and φ^2 differ.

Putting Theorem 5.4.1(i) and Lemma 5.1.1 together gives:

Proposition 5.4.2 (P’s pointers are superfluous for safe terms). *In the game semantics of safe lambda-terms, pointers emanating from P-moves are unnecessary: they are uniquely recoverable from the underlying sequences of moves and from O-moves’ pointers.*

Example 5.4.2. If justification pointers are omitted then the denotations of the two Kierstead terms $M_1 \equiv \lambda f.f(\lambda x.f(\lambda y.y))$ and $M_2 \equiv \lambda f.f(\lambda x.f(\lambda y.x))$ from Example 3.1.1 are not distinguishable. In the safe lambda calculus this ambiguity disappears since M_1 is safe whereas M_2 is not (The free variable x in the subterm $f(\lambda y.x)$, has the same order as y but it is not abstracted together with y).

In fact, as the last example highlights, pointers are superfluous at order 3 for safe terms whether from P-moves or O-moves. This is because for question moves in the first two levels of an arena (initial moves being at level 0), the associated pointers are uniquely recoverable thanks to the visibility condition. At the third level, the question moves are all P-moves therefore their associated pointers are uniquely recoverable by P-incremental justification. This is not true anymore at order 4: Take the safe term $\psi : (((o^4, o^3), o^2), o^1) \vdash_s \psi(\lambda\varphi.\varphi a) : o^0$ for some constant $a : o$, where $\varphi : (o, o)$. Its strategy denotation contains plays whose underlying sequence of moves is $q_0 q_1 q_2 q_3 q_2 q_3 q_4$. Since q_4 is an O-move, it is not constrained by P-incremental justification and thus it can point to any of the two occurrences of q_3 .³

5.5 Safe PCF

We now extend the game-semantic characterization to Safe PCF.

We have already established the correspondence between almost safety and incremental binding in the general setting of an applied simply typed lambda calculus without recursion (Proposition 5.3.1). PCF_1 can be cast into this setting by considering \perp_A as ordinary constants: in the computation tree of a PCF_1 term, subterms of the form Ω_A are represented by the single constant node \perp_A . In full PCF, however, a difficulty arises as computation trees are potentially infinite due to the presence of the Y combinator. Nevertheless the result still holds:

Proposition 5.5.1 (Almost safety and incrementally-binding). *Let $\Gamma \vdash M : A$ be a PCF term.*

- (i) *If $\Gamma \vdash M : A$ is almost safe then $\tau(\Gamma \vdash M : A)$ is incrementally-bound ;*
- (ii) *conversely, if $\tau(\Gamma \vdash M : A)$ is incrementally-bound then the η -normal form of $\Gamma \vdash M : A$ is almost safe if M is open and safe if M is closed.*

Proof. (i) Let M be an almost safe PCF term and i denote the number of occurrences of the Y combinator in M . We first prove by induction on i that for any $k \in \omega$, the k th approximants to M , denoted M_k , is almost safe. The base case $i = 0$ is trivial: $M_k = M$. Step case: $i > 0$. Let $Y_A N$ be a subterm of M . Since M is almost safe, N is also safe. The number of occurrences of the Y combinator in N is smaller than i therefore by the induction hypothesis N_k is safe. Consequently the term $Y_A^k N_k = \underbrace{N_k(\dots(N_k \Omega)\dots)}_{k \text{ times}}$ is also safe and by compositionality so is M_k .

The result holds for PCF_1 terms, thus since M_k is a safe PCF_1 term, $\tau(M_k)$ is incrementally-bound. Now let z be a variable node in $\tau(M) = \bigcup_{k \in \omega} \tau(M_k)$. There exists $k \in \omega$ such that z belongs to $\tau(M_k) \subseteq \tau(M)$. If we write r_k to denote the root of the tree $\tau(M_k)$ then the path $[r_k, z]$ in $\tau(M_k)$ is equal to the path $[r, z]$ in $\tau(M)$. Hence, since z is incrementally-bound in $\tau(M_k)$, it is also incrementally-bound in $\tau(M)$.

³More generally, a P-incrementally justified strategy can contain plays that are not “O-incrementally justified” since it must take into account any possible strategy incarnating its context, including those that are not P-incrementally justified. For instance in the given example, there is one version of the play that is not O-incrementally justified (the one where q_4 points to the first occurrence of q_3). This play is involved in the strategy composition $\llbracket \vdash_{st} M_2 : (((o, o), o), o) \rrbracket; \llbracket \psi : (((o, o), o), o) \vdash_{st} \psi(\lambda\varphi.\varphi a) : o \rrbracket$ where M_2 denotes the unsafe Kierstead term.

(ii) Suppose that the term is not almost safe then necessarily one of its approximant is not almost safe either. Since the result holds for any PCF_1 term, the computation tree of the approximant is not incrementally-bound. But the computation tree of M contains the computation tree of its approximant, therefore it is not incrementally-bound. \square

Hence we obtain the following characterization of almost safety by P-incrementally justified strategies:

Theorem 5.5.1 (Characterization Theorem for Safe PCF). *Let $\Gamma \vdash M : A$ be a PCF term. Then:*

- (i) *If M is almost safe then $\llbracket \Gamma \vdash M : A \rrbracket$ is P-incrementally justified.*
- (ii) *If $\llbracket \Gamma \vdash M : A \rrbracket$ is P-incrementally justified then $\eta_{\text{nf}}(\beta_{\text{nf}}(M))^*$ is almost safe if M is open, and safe if M is closed.*

Proof. (i) Let M be an almost safe term and M^∞ be the β -normal form of M . Since almost-safety is preserved by the small-step reduction of PCF, M^∞ is also almost-safe and by Proposition 5.5.1, $\tau(M^\infty)$ is incrementally-bound. By Corollary 5.3.3(i), $\llbracket M^\infty \rrbracket$ is P-incrementally justified and by soundness of the game denotation, $\llbracket M^\infty \rrbracket = \llbracket M \rrbracket$, thus $\llbracket M \rrbracket$ is P-incrementally justified.

(ii) Let M be PCF term with a P-incrementally justified denotation. By Corollary 5.3.3(ii), $\tau(\beta_{\text{nf}}(M))^* = \tau(\eta_{\text{nf}}(\beta_{\text{nf}}(M))^*)$ is incrementally-bound. Hence by Proposition 5.5.1(ii), if M is closed then $\eta_{\text{nf}}(\beta_{\text{nf}}(M))^*$ is safe and almost safe otherwise. \square

Consequently, P-pointers are superfluous in the game denotation of safe PCF terms: pointers emanating from P-moves are uniquely recoverable.

Example 5.5.1 (Counter-example). The use of dead-code elimination in the second part of the theorem is crucial. Take for instance the closed PCF term:

$$M \equiv \lambda f^{((\text{exp}, \text{exp}), \text{exp})} x^{\text{exp}} y^{\text{exp}}. f(\lambda z^{\text{exp}}. \text{cond}(\text{succ } x) yz) .$$

This term is in β -normal form (the conditional operator cannot be reduced since the value of x is undetermined). The η -long β -normal form of M is therefore M itself which is unsafe. But since $\text{succ } x$ will always evaluate to a positive integer, the first branch of the conditional operator will never be evaluated. Hence M is observationally equivalent to the safe term $N \equiv \lambda fxy. f(\lambda z. z)$ which by the Full Abstraction theorem implies that they have the same denotation. But since N is safe, by the first part of the theorem, we have that $\llbracket M \rrbracket$ is P-incrementally justified.

Such counter-example arises because the conditional operator of PCF permits us to construct beta-normal terms containing “dead code” (*i.e.*, some subterm that will never be evaluated for any value of M ’s parameters). In the example above, the dead code consists of the subterm y . In general, if the dead code part of the computation tree contains a variable that is not incrementally bound then the resulting term will be unsafe even if the rest of the tree is incrementally bound. In our example, it is possible to turn M into the equivalent safe term N by eliminating the dead code from M .

5.6 Safe Idealized Algol

The argument used in the previous section for safe PCF can be reused identically for safe IA (as defined in Sec. 3.5.2.2). Hence we have:

Theorem 5.6.1 (Characterization Theorem for Safe IA). *Let $\Gamma \vdash M : A$ be a IA term. Then:*

- (i) *If M is almost safe then $\llbracket \Gamma \vdash M : A \rrbracket$ is P-incrementally justified.*
- (ii) *If $\llbracket \Gamma \vdash M : A \rrbracket$ is P-incrementally justified then $\eta_{\text{nf}}(\beta_{\text{nf}}(M))^*$ is almost safe if M is open, and safe if M is closed.*

This shows that P-pointers are superfluous for safe IA terms. Since unsafety only appears at order 3, this theorem implies the well-known result that pointers are uniquely recoverable for IA_2 terms. This suggests potential applications in Algorithmic Game Semantics: Ghica and McCusker were able to show that the game denotation of IA_2 terms can be characterized by (extended) regular expressions, thus giving a decision procedure for observational equivalence in this fragment [GM00]. Can we achieve a result for higher-order fragment of safe IA? We will investigate this question in the next chapter.

5.7 Towards a game model of safe PCF

5.7.1 Definability

Recall (Sec. 2.3.5.5) that PCF_c denotes the language obtained by extending PCF with the case_k construct. The case_k construct is the obvious generalization of the conditional operator cond to $k \in \mathbb{N}$ branches instead of 2. We call safe PCF_c the corresponding extension of safe PCF. Clearly, all the results obtained so far concerning safe PCF also hold in safe PCF_c .

The characterization theorem allows us to show the following definability result for safe PCF_c :

Proposition 5.7.1 (Definability for safe PCF_c terms). *Let $\bar{A} = (A_1, \dots, A_i)$ and B be two PCF types for some $i, l \geq 0$ and σ be a well-bracketed innocent P-i.j. strategy with finite view function defined on the game $A_1 \times \dots \times A_i \rightarrow B$. There exists an almost safe PCF_c term $\bar{x} : \bar{A} \vdash M : B$ in η -long normal form such that:*

$$\llbracket \bar{x} : \bar{A} \vdash M_\sigma : B \rrbracket = \sigma$$

and a safe closed PCF_c term $\vdash_s M'_\sigma : (\bar{A}, B)$ in η -long normal form such that:

$$\llbracket \vdash_s M'_\sigma : (\bar{A}, B) \rrbracket \cong \sigma.$$

Proof. By the standard definability result for PCF_c , there is a *finite* term $\bar{x} : \bar{A} \vdash N : B$ such that $\llbracket \bar{x} : \bar{A} \vdash N : B \rrbracket = \sigma$. Take M_σ to be $\eta_{\text{nf}}(\beta_{\text{nf}}(N))^*$. We have $\llbracket \bar{x} : \bar{A} \vdash M_\sigma : B \rrbracket = \llbracket \bar{x} : \bar{A} \vdash N : B \rrbracket = \sigma$ and by Theorem 5.5.1(ii), M_σ is almost safe. For the second part, take M'_σ to be the closure $\lambda \bar{x}. M_\sigma$ of M_σ . \square

Note that because the argument relies on dead code-elimination, which is undecidable, it does not constitute a constructive proof: we know that the term M_σ exists but we do not have an algorithm to compute it.

This result shows that the game model of Safe PCF is **intentionally fully-abstract**: every *compact* strategies (i.e., with finite view function) is definable [AMJ94]. The property that all denotations in the model are definable, including the recursive ones, is called **universality**. Universality was shown for the game model of PCF [AMJ94]. In order to show universality for safe PCF, the “trick” used in the previous proof does not suffice: It is possible to perform dead-code elimination on the infinite term obtained by unfolding the Y-recursion, but the resulting term is a potentially infinite term, and it is not necessarily the unfolding of a “finite” PCF term (with Y combinators). Thus one has to be slightly more subtle to handle recursion. One way around this problem could consist in using a version of the Correspondence Theorem expressed over a finite syntax representation of the term (as described in remark 4.3.1) and to perform dead-code elimination on this representation rather than on its unwinding. We will not investigate this question further as it is not essential to our understanding of the game semantics of safe calculi.

5.7.2 Compositionality

In the next chapter we will give an in depth account of P-i.j. strategies. In particular we will give a semantic argument showing that when suitably restricted, P-i.j. strategies compose. We show here essentially the safe result using a syntactic argument that relies on the definability result from the previous section. The advantage is that the proof is much simpler than the one given in

the next chapter. The disadvantage is that it is slightly less general as it only works for strategies that are denotations of compact PCF terms (*i.e.*, the compact innocent ones) whereas the proof in the next chapter works in the general case.

Let $\bar{A} = (A_1, \dots, A_i)$, $B = (B_1, \dots, B_l, o)$ and $C = (C_1, \dots, C_k, o)$ be three PCF types for some $i \geq 1, l, k \geq 0$.

Problem: Given two compact (with finite view function) innocent well-bracketed and P-incrementally justified strategies $f : A_1 \times \dots \times A_i \rightarrow B$ and $g : B \rightarrow C$. What is a sufficient condition for the composite $f;g$ to be P-incrementally justified?

We tackle the problem syntactically by appealing to the definability result: since f and g are compact innocent, there are two closed safe terms $M_f : (\bar{A}, B)$ and $M_g : B \rightarrow C$ in η -long nf denoted by f and g respectively. Composition is syntactically formulated by the term

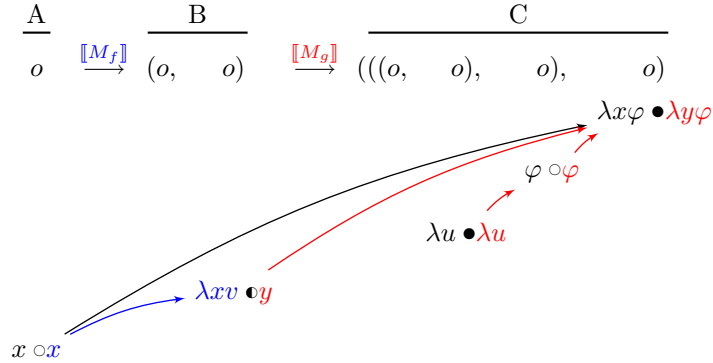
$$M_{f;g} \equiv \lambda \bar{x}. M_g(M_f \bar{x})$$

for some fresh variables $\bar{x} : \bar{A}$, whose denotation is clearly given by $\llbracket M_f \rrbracket; \llbracket M_g \rrbracket = f; g$.

Observe that the safety of M_f and M_g does not imply that of $M_{f;g}$ as the following examples illustrate:

Example 5.7.1. (i) Take $A = o$, $B = (o, o)$, $C = (((o, o), o), o)$, the variables $x, u, v : o$, $y : B$ and $\varphi : ((o, o), o)$ and the Σ -constant $a : o$. Take the two closed safe terms $M_f \equiv \lambda x v. x$: $A \rightarrow B$ and $M_g \equiv \lambda y \varphi. \varphi(\lambda u. y a)$: $B \rightarrow C$. The eta-long beta-nf of $M_{f;g}$ is $\lambda x \varphi. \varphi(\lambda u. x)$ which is unsafe because of the underlined term.

Consequently by Theorem 5.4.1(ii), the strategy $\llbracket M_{f;g} \rrbracket = \llbracket M_f \rrbracket; \llbracket M_g \rrbracket$ is not P-i.j. This shows that P-i.j. strategies do not generally compose. The following diagram illustrates a play that is not P-i.j.:



(ii) A counter-example with $\text{ord } B = \text{ord } C$: Let $A = o$, $B = C = (((o, o), o), o)$ and let $x : A$, $y : B$, $u : o$, $v, \varphi : ((o, o), o)$ and $g : (o, o)$ be variables and $a : o$ be a Σ -constant. Take the two closed safe terms $M_f \equiv \lambda x v. x$ and $M_g \equiv \lambda y \varphi. \varphi(\lambda u. y(\lambda g. a))$. The $\eta\beta$ -nf of $M_{f;g}$ is $\lambda x \varphi. \varphi(\lambda u. x)$ which is unsafe because of the underlined term, so $f;g$ is not P-i.j.

Since M_f and M_g are in η -nf, they can be written:

$$\begin{aligned} \vdash_s M_f &\equiv \lambda x_1^{A_1} \dots x_i^{A_i} \varphi_1^{B_1} \dots \varphi_l^{B_l} . N_f \\ \vdash_s M_g &\equiv \lambda y^{(B_1, \dots, B_l, o)} \phi_1^{C_1} \dots \phi_k^{C_k} . N_g \end{aligned}$$

for some safe ground-type terms N_f and N_g in η -nf. Substituting these two equations in $M_{f;g}$ gives:

$$\begin{aligned} f;g &= \llbracket \lambda \bar{x}. (\lambda \phi_1 \dots \phi_k. N_g) [(M_f \bar{x})/y] \rrbracket \\ &= \llbracket \lambda \bar{x} \phi_1 \dots \phi_k. N_g [(M_f \bar{x})/y] \rrbracket \quad (\text{the } x_j\text{'s and } \phi_j\text{'s can be chosen to be disjoint}). \end{aligned} \quad (5.2)$$

Thus by Theorem 5.5.1, $f;g$ is P-incrementally justified just when $\eta_{\text{nf}}(\beta_{\text{nf}}(N_g[(M_f \bar{x})/y]))^*$ is safe.

A sufficient and necessary condition

Lemma 5.7.1. *Let $\Gamma, y : B \vdash_s M$ be a safe term in η -nf and $\Gamma \vdash R : B$ be an almost safe application. Let N denote the set of nodes of the computation tree of M and \otimes be the root. Then:*

$$\Gamma \vdash_s M[R/y] : A \iff \forall x \in FV(R). \forall y \in N_{fv}. \forall m \in N_{\lambda} \cap [\otimes, y] : \text{ord } m \leq \text{ord } x.$$

Proof. The only cause of unsafety that can be introduced when substituting the almost safe term R for y in M is when some variable free in R becomes not incrementally bound in $\tau(M)$. The right-hand side of the equivalence expresses just this. \square

Applying this lemma with $R \equiv M_f \bar{x}$ and $M \equiv M_g$ gives us a necessary and sufficient condition for $M_g[(M_f \bar{x})/y]$ to be safe, and hence for $f;g$ to be P-i.j. The problem is that this condition is expressed on both M_g and M_f at the same time rather than independently. This is unsatisfactory because it does not give rise to a categorical notion of compositionality: two morphisms should be composable as soon as the domain of one matches with the codomain of the other.

A sufficient condition The solution consists in restricting the P-i.j. strategies to a smaller class of composable strategies.

Lemma 5.7.2. *If $\text{ord } A_i \geq \text{ord } B$ for all $1 \leq i \leq n$ then $f;g$ is P-incrementally justified.*

Proof. For all $1 \leq i \leq n$ we have $\text{ord } x_i = \text{ord } A_i \geq \text{ord } B = \text{ord } (M_f \bar{x})$ thus we can use the application rule of the safe lambda calculus to form the safe term $\bar{x} : \bar{A} \vdash_s M_f \bar{x}$. The substitution lemma then shows that $M_g[(M_f \bar{x})/y]$ is safe which by Eq. 5.2 implies that $f;g$ is P-i.j. \square

Strategies verifying this condition are the *closed P-incrementally justified strategies*. This property will be studied in depth in Sec. 6.2.4.

REMARK 5.7.1

1. The condition is not necessary: Take $A = o$, $B = (o, o)$, $C = (o, o)$ and consider the two safe terms $M_f \equiv \lambda x^A u^o.u$ and $M_g \equiv \lambda y^B.y a$ for some constant $a : o$. Then we have $M_{f;g} =_{\beta} \lambda x.a$ which is safe hence $f;g$ is P-i.j. although $\text{ord } A < \text{ord } B$.
2. In general type homogeneity is not preserved after composition. For instance the types $o \rightarrow (o \rightarrow o)$ and $(o \rightarrow o) \rightarrow ((o \rightarrow o) \rightarrow o)$ are homogeneous but $o \rightarrow ((o \rightarrow o) \rightarrow o)$ is not. Incidentally, the condition of Lemma 5.7.2 turns out to be a sufficient condition for type-homogeneity to compose: for instance if $A \rightarrow B$ and $B \rightarrow C$ are homogeneous simple types and $\text{ord } A \geq \text{ord } B$ then $A \rightarrow C$ is homogeneous.

5.7.3 Full abstraction

In Chapter 2 we have presented the well-known result that the standard game models of PCF is fully abstract [AMJ94, HO00, Nic94]: two PCF terms are observationally equivalent if and only if they have the same denotations. Since safe PCF is a fragment of PCF this statement also holds for safe PCF terms: two safe PCF terms are observationally equivalent *with respect to PCF contexts* (not necessarily safe) if and only if they have the same game denotation.

A natural question to ask is whether there exists a fully abstract model with *respect to safe contexts* only. Since safe PCF terms are denoted by P-incrementally justified strategies, it is reasonable to think that O-moves also need to be constrained by a symmetrical notion of “O-incremental justification” corresponding to the requirement that contexts are safe.

The definability result shown for safe PCF is a first step towards full-abstraction. This problem will be studied in Chapter 6.

Chapter 6

Models of Safe Applied Lambda Calculi



This chapter aims to formally define what is a model of the safe lambda calculus and its various extensions. We present a categorical interpretation of the safe lambda calculus in the same vein as the characterization of the lambda calculus by Cartesian Closed Categories. We then provide such a model by means of game semantics and show that it is fully-abstract when observational equivalence is defined with respect to safe contexts. We conclude the chapter by examining the model from an Algorithmic game semantic point of view: we consider the problem of observational equivalence for finitary fragments of safe IA and show that up to order 3, the complexity of deciding observational equivalence is essentially the same as for unrestricted IA terms. We then give a version of the complete play Characterization Theorem for safe terms: we show that two safe terms are observationally equivalent if and only if the sets of complete O-incrementally justified plays of the denotations are equal. This result leads us to conjecture that observational equivalence is decidable for safe IA_4 .

6.1 Categorical model

It is well-known [Lam86] that cartesian closed categories (a category with a terminal object, finite product and exponential), CCCs for short, capture the notion of model of typed lambda calculi: any extensional model of the simply typed lambda calculus is a CCC. Conversely, any CCC induces a typed-lambda calculus. *What is the categorical interpretation of the safe lambda calculus?* This section introduces *incremental closed categories* and shows that they capture models of safe lambda calculi.

6.1.1 Safe lambda calculus with product

The safe lambda calculus defined in Chapter 3 does not have a product. It is fairly easy to add it to the language. The type grammar is given by:

$$T ::= B \mid T \rightarrow T \mid T \times T$$

for some set B of base types. The typing system of the safe lambda calculus is then extended with three rules corresponding to pairing, first projection and second projection (respectively (\times) , (π_1) and (π_2) in Table 6.1). Although this suffices to add product to the safe lambda calculus, we will impose a further restriction. Indeed, consider the following examples:

$$\begin{aligned} x : (o \rightarrow o) \times o \vdash_{\text{st}} \lambda z^o. (\pi_2 x) : (o \rightarrow (o \rightarrow o)) &\equiv M_1 \\ x_1 : (o \rightarrow o), x_2 : o \vdash_s \lambda z^o. \underline{x_2} : (o \rightarrow (o \rightarrow o)) &\equiv M_2 \end{aligned}$$

Although these are two equivalent beta-normal terms, only M_2 is safe. Indeed, the side condition of the abstraction rule only requires that the *variables* in the context have order greater than the order of the term, therefore M_2 is unsafe because of the variable x_2 . In M_1 , however, x_1 and x_2 are combined into a single variable, this has the effect of increasing the order of the variable and therefore the side-condition holds.

In the categorical model of the simply typed lambda calculus, a term-in-context $\Gamma \vdash M : T$ is modeled by a morphism $[\Gamma] \rightarrow [T]$ where the context Γ is identified with the product of the types of the variables in the context: if the context variables are X_1, \dots, X_n then Γ is identified with $X_1 \times \dots \times X_n$. Thus two contexts $x : A, y : B$ and $xy : A \times B$ will be denoted by the same object in the category. Because variables in the context can be “combined”, there is no way to tell just by looking at the type Γ which subtypes come from which variable. Consequently the basic property of the safe lambda calculus—that all the variables in the context have order greater than the order of the term—cannot be expressed in the standard categorical model. For this reason we modify slightly the side-condition of the abstraction and application rules to enforce a property stronger than the usual basic property of the safe lambda calculus: instead of requiring that all variables in the context have order greater than the order of the term, we require that the order of *any prime sub-type of any variable* in the context has order greater than that of the term, where the set of **prime sub-types** of a type A , written $Pr(A)$, is given by:

$$\begin{aligned} Pr(B) &= \{B\} && \text{if } B \text{ is a base type,} \\ Pr(A \rightarrow B) &= \{A \rightarrow B\} \\ Pr(A \times B) &= Pr(A) \cup Pr(B) . \end{aligned}$$

We then define the relation \geq on types as follows:

$$A \geq B \stackrel{\text{def}}{=} \forall A' \in Pr(A). \text{ord } A' \geq \text{ord } B .$$

Thus for any context Γ and type B we have:

$$\Gamma \geq B \iff \forall x : A \in \Gamma. \forall A' \in Pr(A). \text{ord } A' \geq \text{ord } B .$$

We now replace the side-condition in the abstraction and application rules by “ $\Gamma \geq B$ ” where B denotes the type of the term being formed and Γ its context.

Definition 6.1.1. The **safe lambda calculus with product**, or safe Λ^\times , for short, over a typed-alphabet Ξ of constants is given by induction over the rules of Table 6.1. The differences with the rules of the safe lambda calculus without product are framed.

Example 6.1.1. The terms M_1 and M_2 given above are both unsafe.

It is easy to see that the basic property of the safe lambda calculus still holds—the free variables of a term have order greater than the order of the term itself—and therefore all the basic results showed in Chapter 3 also hold (No-variable-capture lemma, safety is preserved by safe β reduction, ...).

We call **typed-calculus** any applied simply typed lambda calculus with product with a stock of constants and function symbols together with an operational semantics for function symbols given by means of a set of reduction rules. We define the **safe fragment** of a typed-calculus as the system obtained by replacing the abstraction and application rules by the rules (app), (app_{as}), (abs) and (δ) from Table 6.1. A language that is the safe fragment of some typed-lambda calculus is called a **safe typed-calculus**.

The **long safe fragment** of a type-calculus is the subclass of the safe fragment consisting of terms-in-context that are typable without using the rule (app_{as}). (See Definition 3.1.8.)

$$\begin{array}{c}
(\text{var}) \frac{}{x : A \vdash_s x : A} \quad (\text{const}) \frac{}{\vdash_s f : A} f \in \Xi \quad (\text{wk}) \frac{\Gamma \vdash_s s : A}{\Delta \vdash_s s : A} \quad \Gamma \subset \Delta \quad (\delta) \frac{\Gamma \vdash_s M : A}{\Gamma \Vdash_{\text{app}} M : A} \\
\\
\boxed{(\times) \frac{\Gamma \vdash_s s : A \quad \Gamma \vdash_s t : B}{\Gamma \vdash_s \langle s, t \rangle : A \times B} \quad (\pi_1) \frac{\Gamma \vdash_s s : A \times B}{\Gamma \vdash_s \pi_1 s : A} \quad (\pi_2) \frac{\Gamma \vdash_s s : A \times B}{\Gamma \vdash_s \pi_2 s : B}} \\
\\
(\text{app}_{\text{as}}) \frac{\Gamma \vdash_s s : (A_1, \dots, A_n, B) \quad \Gamma \vdash_s t_1 : A_1 \quad \dots \quad \Gamma \vdash_s t_n : A_n}{\Gamma \Vdash_{\text{app}} s \ t_1 \dots t_n : B} \\
\\
(\text{app}) \frac{\Gamma \vdash_s s : (A_1, \dots, A_n, B) \quad \Gamma \vdash_s t_1 : A_1 \quad \dots \quad \Gamma \vdash_s t_n : A_n}{\Gamma \vdash_s s \ t_1 \dots t_n : B} \quad \boxed{\Gamma \geq B} \\
\\
(\text{abs}) \frac{\Gamma, x_1 : A_1, \dots, x_n : A_n \Vdash_{\text{app}} s : B}{\Gamma \vdash_s \lambda x_1 \dots x_n. s : (A_1, \dots, A_n, B)} \quad \boxed{\Gamma \geq (A_1, \dots, A_n, B)}
\end{array}$$

Table 6.1: The safe lambda calculus with product.

Alternative definition Our definition of the safe lambda calculus with product conveys the syntactic notion of safety appropriately but it is not semantically consistent in the sense that there exists pairs of terms that are denoted by the same morphism in the categorical model of the simply typed lambda calculus but such that one is safe and the other unsafe. For instance the two simply typed terms:

$$\begin{aligned}
x : (o \rightarrow o) \times o \vdash_{\text{st}} \lambda z^o. (\pi_1 x) : (o \rightarrow (o \rightarrow o)) &\equiv N_1 \\
x_1 : (o \rightarrow o), x_2 : o \vdash_s \lambda z^o. x_1 : (o \rightarrow (o \rightarrow o)) &\equiv N_2
\end{aligned}$$

are denoted by the same morphism in the categorical model, but N_1 is unsafe whereas N_2 is safe. (This is because in N_1 , the variable x has to be introduced first in the derivation tree, whereas in N_2 , x_1 has to be introduced first but x_2 can be added to the context at the end of the derivation using the weakening rule.)

We could define an alternative notion of safe lambda calculus with product where this mismatch does not happen. One way is to impose that for any context-variable of type $A \times B$ the equality $\text{ord } A = \text{ord } B$ holds. Another solution is to forbid the use of variables of product type and only allow product types for terms created with the pairing rule. But these two approaches are rather restrictive. A better approach consists in changing the system to allow the formation of terms like N_2 . This can be done by adding a new kind of weakening rule that alters the type of context-variables rather than adding new variables to the context:

$$(\text{wk}^\times) \frac{\Gamma, x : A \vdash_s s : C}{\Gamma, x : A \times B \vdash_s s[(\pi_1 x)/x] : C}$$

Semantically, this rule is equivalent to the weakening rule because in the categorical model of the simply typed lambda calculus, if s is denoted by a morphism $\llbracket s \rrbracket : \Gamma \times A \rightarrow C$ then $\Gamma, x : A \times B \vdash_{\text{st}} s[(\pi_1 x)/x] : C$ and $\Gamma, x : A, y : B \vdash_{\text{st}} s[(\pi_1 x)/x] : C$ are denoted by the morphisms $(id_\Gamma \times \pi_1^{A \times B}); \llbracket s \rrbracket$ and $\pi_1^{(\Gamma \times A) \times B}; \llbracket s \rrbracket$. These two denotations are the same since $id_\Gamma \times \pi_1^{A \times B} = \langle \pi_1^{\Gamma \times (A \times B)}; id_\Gamma, \pi_2^{\Gamma \times (A \times B)}; \pi_1^{A \times B} \rangle$, which by associativity of the product is isomorphic to $\langle \pi_1^{(\Gamma \times A) \times B}; \pi_1^{\Gamma \times A}, \pi_2^{(\Gamma \times A) \times B}; \pi_2^{\Gamma \times A} \rangle = \pi_1^{(\Gamma \times A) \times B}$.

Example 6.1.2. With the addition of this rule to the system, both N_1 and N_2 are typable.

Again it is easy to see that the basic property of the safe lambda calculus still holds and therefore all the basic results showed in Chapter 3 also hold. Moreover, any term that is typable with these rules is equivalent to some term typable with the rules of the safe lambda calculus with product. Thus any model of the safe lambda calculus with product is also a model of this calculus.

6.1.2 Incremental closed category

We first recall some basic categorical notions and fix some notations.

Basic definitions

A **category** \mathbf{C} is given by a class $\text{Obj}(\mathbf{C})$ of objects and a class $\text{Hom}(\mathbf{C})$ of morphisms between objects: for each pair of objects A, B , a set of morphisms $\mathbf{C}(A, B)$, written $f : A \rightarrow B$, where A is the domain and B is the codomain. Further for any three objects A, B and C , and morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$ there is a composite morphism written $f;g$ or $g \circ f$ such that the composition operation is associative and for each object A there is a morphism id_A that is the identity for composition.

A **subcategory** of a category \mathbf{C} is a category whose objects and morphisms are objects and morphisms of \mathbf{C} . It is a *lluf* subcategory if it contains all the objects of \mathbf{C} .

An object I is **terminal** if for every object A there is a unique morphism from A to I .

A category has **products** if for any two objects A and B there is an object $A \times B$ and two morphisms π_1, π_2 mapping $A \times B$ to A and B respectively such that for any morphisms $f : C \rightarrow A$, $g : C \rightarrow B$, there is a unique morphism $\langle f, g \rangle : C \rightarrow A \times B$, called the **pairing** of f and g such that $\pi_2 \circ \langle f, g \rangle = g$ and $\pi_1 \circ \langle f, g \rangle = f$, where $f \circ g$ denotes the composition of g with f in the category.

A category has **exponential** if for any two objects B and C there is a distinguished object C^B and a morphism $ev_{B,C} : (C^B \times B) \rightarrow C$ such that for any object A and morphism $f : (A \times B) \rightarrow C$ there is a unique morphism $\Lambda(f) : A \rightarrow C^B$ such that the following diagram commutes:

$$\begin{array}{ccc} A \times B & & \\ \downarrow \Lambda(f) \times id_B & \searrow f & \\ C^B \times B & \xrightarrow{ev_{B,C}} & C \end{array}$$

Definition 6.1.2. A **cartesian closed category**, CCC for short, is a category with a terminal object, binary products and exponentials.

Incremental closed category

Let \mathbf{C} be a CCC. We define an *order* function $\text{ord} : \text{Obj}(\mathbf{C}) \rightarrow \mathbb{N} \cup \{-1\}$ and a function $\text{dro} : \text{Obj}(\mathbf{C}) \rightarrow \mathbb{N} \cup \{-1\}$ on objects as follows:

- $\text{ord}(I) = \text{dro}(I) = -1$,
- $\text{ord}(A \times B) = \max(\text{ord } A, \text{ord } B)$ and $\text{dro}(A \times B) = \min(\text{dro}(A), \text{dro}(B))$,
- $\text{ord}(B^A) = \text{dro}(B^A) = \max(1 + \text{ord } A, \text{ord } B)$,
- $\text{ord } A = \text{dro}(A) = 0$ for any other object A .

Clearly we have $\text{ord} \geq \text{dro}$ hence:

Lemma 6.1.1. For any objects A, B and C of a CCC, if $\text{dro}(A) \geq \text{ord}(B)$ and $\text{dro}(B) \geq \text{ord}(C)$ then $\text{dro}(A) \geq \text{ord}(C)$.

We say that a morphism $f : A \rightarrow B$ is **incremental** if we have $\delta(A_1) < \text{ord}(B)$. We say that a subcategory of a CCC is **incremental** if all the morphism of the subcategory are incremental.

Definition 6.1.3. Let \mathbf{C} be a cartesian closed category. An **incremental closed subcategory**, sub-ICC for short, is a lluf subcategory \mathbf{I} of \mathbf{C} such that:

1. it contains all the projections: for all objects C_1 and C_2 , $\pi_1 : C_1 \times C_2 \rightarrow C_1$ and $\pi_2 : C_1 \times C_2 \rightarrow C_2$ are in $\text{Hom}(\mathbf{I})$;
2. it is closed under pairing: if $f : C \rightarrow A$ and $g : C \rightarrow B$ are in $\text{Hom}(\mathbf{I})$ then so is $\langle f, g \rangle$;

3. it contains all the incremental evaluation morphisms: for any objects B and C such that $\text{dro}(B) \geq \text{ord}(C)$, $\text{ev}_{B,C} : (C^B \times B) \rightarrow C$ is in $\text{Hom}(\mathbf{I})$;
4. it is closed under incremental currying: if $f : (A \times B) \rightarrow C \in \text{Hom}(\mathbf{I})$ with $\text{dro}(A) \geq \text{ord}(C^B)$ then $\Lambda(f) : A \rightarrow C^B \in \text{Hom}(\mathbf{I})$;
5. all morphisms are incremental modulo weakening: for any morphism $f : A \rightarrow B$, either f is incremental, or $A = A_1 \times A_2$ and $f = \pi_1; g$ for some incremental morphism $g : A_1 \rightarrow B$.

An **incremental closed category**, ICC for short, is a category that is isomorphic to some sub-ICC of a CCC.

Let \mathbf{C} be a CCC. The **canonical sub-ICC** of \mathbf{C} , written $\text{subICC}(\mathbf{C})$, is the subcategory \mathbf{I} obtained by keeping only the morphisms that are incremental modulo weakening. Formally for any objects A, B :

$$\begin{aligned} \mathbf{I}(A, B) &= \mathbf{C}(A, B) && \text{if } \text{dro}(A) \geq \text{ord}(B); \\ \mathbf{I}(A, B) &= \emptyset && \text{if } \text{dro}(A) < \text{ord}(B) \text{ and } A \text{ is not a product;} \\ \mathbf{I}(A_1 \times A_2, B) &= \{\pi_1; f \mid f \in \mathbf{I}(A_1, B)\} && \text{otherwise.} \end{aligned}$$

Proposition 6.1.1. *The canonical sub-ICC is a well-defined sub-ICC.*

Proof. Let \mathbf{C} denote the CCC and \mathbf{I} the canonical sub-ICC. We first show that \mathbf{I} is indeed a subcategory: The identity morphisms id_A are all incremental therefore they are all in $\text{Hom}(\mathbf{I})$, and the class of morphism is closed under composition. Indeed take two morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$:

- if f and g are incremental then by Lemma 6.1.1, $f; g$ is incremental;
- if $f = \pi_1; f'$ and f' and g are incremental for some projection π_1 then $f; g = (\pi_1; f'); g = \pi_1; (f'; g)$ by associativity. Since f' and g are incremental, so is $f'; g$ therefore $f; g$ is incremental modulo weakening;
- if $g = \pi_1; g'$ and f and g' are incremental for some projection π_1 then we have $B = B_1 \times B_2$ and $\text{dro}(A) \geq \text{ord}(B) \geq \text{ord}(B_1) \geq \text{dro}(C) \geq \text{ord}(C)$, therefore $f; g : A \rightarrow C$ is incremental;
- if $f = \pi_1; f'$ and $g = \pi_1; g'$ where f' and g' are incremental then the last two points show that $f; g$ is incremental modulo weakening.

Hence \mathbf{I} is a subcategory. Clearly \mathbf{I} contains the projections (a projection $\pi_1 : C_1 \times C_2 \rightarrow C_1$ that is not incremental can always be written $\pi_1 = \pi_1; \text{id}_{C_1}$ where id_{C_1} is incremental), and it is closed under pairing. Also by definition it contains all the incremental evaluation morphism from \mathbf{C} , it is closed under incremental currying, and all morphisms in the category are incremental modulo weakening. Hence \mathbf{I} is *incremental closed*. \square

An object A of a CCC is said to be **homogeneous** if

- A is not a product, nor an exponential;
- or $A = B \times C$ where B and C are homogeneous and $\text{ord } B \geq \text{ord } C$;
- or $A = B \rightarrow C$ where B and C are homogeneous and $\text{ord } B \geq \text{ord } C - 1$.

An **homogeneous incremental category** is a sub-category of an ICC such that its objects are the homogeneous objects of the ICC and its morphisms are the incremental morphisms (but not those that are only incremental modulo weakening).

Order-enrichment

In order to model applied lambda calculi with recursion, one needs to impose further requirement on the category. A condition called *rationality* [AM99] is sufficient for a CCC to interpret PCF. We now adapt this definition to ICCs.

First we reproduce here the definition of rationality [AM99]. A *pointed poset* is a partially ordered set with a least element. A category \mathbf{C} is **pointed-poset** enriched (ppo-enriched) if

- Every hom-set has a pointed poset structure $(\mathbf{C}(A, B), \sqsubseteq_{A,B}, \perp_{A,B})$

- Composition, pairing and currying are monotone.
- Composition is *left-strict*: for all $f : A \rightarrow B$, $\perp_{B,C} \circ f = \perp_{A,C}$.

A category \mathbf{C} is **rational** if it is ppo-enriched and for all $f : A \times B \rightarrow B$, the chain defined by $f^{(0)} = \perp_{A,B}$, $f^{(k+1)} = f \circ \langle id_A, f^{(k)} \rangle$ has a least upper bound denoted by f^∇ such that for all $g : C \rightarrow A$, $h : B \rightarrow D$, $g \circ f^\nabla \circ h = \bigcup_{k \in \omega} g \circ f^{(k)} \circ h$.

We extend this definition to ICCs as follows:

Definition 6.1.4. An ICC is **rational** if it is isomorphic to a sub-ICC \mathbf{I} of a rational CCC such that \mathbf{I} is complete with respect to $(\cdot)^\nabla$ (i.e., if $f : A \times B \rightarrow B$ is a morphism of the subcategory then so is f^∇).

6.1.3 Categorical semantics

Consider a typed-lambda calculus extended with a set of constants and function symbols together with a set of reduction rules giving the operational interpretation of these functions. A **model** of a type-lambda calculus in a cartesian closed category is specified by giving:

- For every ground type T an object $\llbracket T \rrbracket$ of the category. This suffices to interpret any simple type T as an object $\llbracket T \rrbracket$ using products and exponentials;
- for every constant k of type T a morphism $\llbracket K \rrbracket$ of type $\llbracket T \rrbracket$;
- for every function symbol f of type $A_1 \times \dots \times A_n \rightarrow B$, a morphism $\llbracket f \rrbracket$ of type $\llbracket A_1 \rrbracket \times \dots \times \llbracket A_n \rrbracket \rightarrow \llbracket B \rrbracket$.

It is then possible to specify the interpretation of any term-in-context $\Gamma \vdash M : T$ by induction on the structure of the term [Cro93]. The **model** is said to be sound if whenever M reduces to N with the small-step semantics of the language then M and N have the same denotation in the model.

Proposition 6.1.2 (Models of safe typed lambda calculus). *Let \mathcal{L} be a type-lambda calculus. Suppose that a CCC \mathbf{C} provides a sound model of \mathcal{L} , then the canonical sub-ICC \mathbf{I} of \mathbf{C} provides a sound model of the safe fragment of \mathcal{L} .*

Proof. The interpretation $\llbracket \cdot \rrbracket$ of the safe lambda calculus with product in \mathbf{I} is induced by the standard interpretation in the CCC: Ground types are interpreted as objects of the category, this suffices to interpret any simple type T as an object $\llbracket T \rrbracket$ using products and exponentials. A closed term of type T is interpreted by a morphism $I \rightarrow \llbracket T \rrbracket$, and an open term of type T is interpreted by a morphism from the denotation of the type of its free variables to $\llbracket T \rrbracket$.

We show that for any safe term M , its denotation $\llbracket M \rrbracket_{\mathbf{C}}$ in \mathbf{C} is also a morphism of the subcategory \mathbf{I} . Since the model \mathbf{C} is sound, M has the same denotation as its eta-long normal form therefore we can assume w.l.o.g. that M is eta-long normal. We show the result by induction on the structure of M . We do not have to consider the rule (**app**_{as}) because it is not required to type η -long normal terms. The (**var**) axiom is interpreted by the identity morphisms which all belong to the ICC. The rules (\times) , (π_1) and (π_2) are interpreted by pairing and projections. The weakening rule (**wk**) is interpreted by composition with the projection morphism π_1 . For the rule (**app**), the term formed is of ground type (since we work with eta-long normal form) so we have $\llbracket s t_1 \dots t_n : o \rrbracket = \langle \llbracket s \rrbracket, \llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket \rangle \circ ev_{(A_1 \times \dots \times A_n), o}$ so we can conclude using the I.H. on the fact that the evaluation map $ev_{(A_1 \times \dots \times A_n), o}$ belongs to the ICC. Rule (**abs**): let $f : \Gamma \times (A_1 \times \dots \times A_n) \rightarrow T$ be the denotation of the premise. The term formed is denoted by the curried morphism $\Lambda(f) : \Gamma \rightarrow T^{(A_1 \times \dots \times A_n)}$. The side condition ensures that this morphism is incremental closed and therefore it belongs to the ICC.

Hence for any safe term M , we can define its interpretation $\llbracket M \rrbracket_{\mathbf{I}}$ in \mathbf{I} to be its interpretation in \mathbf{C} : $\llbracket M \rrbracket_{\mathbf{I}} \stackrel{\text{def}}{=} \llbracket M \rrbracket_{\mathbf{C}}$. The soundness of the ICC model follows from that of the CCC model. \square

Example 6.1.3 (Model of safe PCF). Any rational CCC in which we have fixed an interpretation for base types, PCF constants and function symbols provides a sound model of PCF [AMJ94]. Therefore any rational ICC provides a sound model of safe PCF. The interpretation of safe PCF in the ICC coincides with its interpretation in the corresponding CCC [AMJ94]: each constant

and first-order function of PCF of type T is interpreted by some morphism $c : I \rightarrow \llbracket T \rrbracket$, and because the category is rational, the Y-combinator Y_A for any object A can be interpreted by the morphism $\Theta_A^\nabla : I \rightarrow A^{A^A}$ where

$$\Theta_A = \llbracket F : (A \rightarrow A) \rightarrow A \vdash \lambda f^{A \rightarrow A}. f(Ff) : (A \rightarrow A) \rightarrow A \rrbracket .$$

6.1.4 Quotiented category

Let \mathbf{C} be a rational CCC. A *precongruence* \lesssim on \mathbf{C} is defined as a family of preorders $\lesssim_{A,B} \subseteq \mathbf{C}(A,B) \times \mathbf{C}(A,B)$ such that $\sqsubseteq_{A,B} \subseteq \lesssim_{A,B}$, composition, pairing, currying are \lesssim -monotonous, and the preorders verify a continuity property [AMJ94]. Given a precongruence, the quotiented category \mathbf{C}/\lesssim is defined as follows: the objects are the same as in \mathbf{C} , and a morphism in \mathbf{C}/\lesssim (A,B) is an equivalence class $[f]$ of $\mathbf{C}(A,B)$ modulo the equivalence relation induced by $\lesssim_{A,B}$. A partial ordering $\leq_{A,B}$ on \mathbf{C}/\lesssim (A,B) can then be defined as follows:

$$[f] \leq_{A,B} [g] \iff f \lesssim_{A,B} g .$$

Lemma 6.1.2 ([AMJ94]). *If \lesssim is a precongruence on a rational CCC \mathbf{C} then \mathbf{C}/\lesssim is a rational CCC.*

The notion of quotient category extends naturally to ICCs. A precongruence \lesssim on an ICC \mathbf{I} is defined similarly as CCC precongruences except that monotonicity is required for *incremental* currying only. This then gives rise to the notion of quotiented category \mathbf{I}/\lesssim .

Lemma 6.1.3. *Let \mathbf{I} be an ICC subcategory of a CCC \mathbf{C} , and let \lesssim be a precongruence on \mathbf{C} . Then:*

- i. \mathbf{I}/\lesssim is (isomorphic to) a sub-ICC of \mathbf{C}/\lesssim ;
- ii. If \mathbf{I} is a rational sub-ICC of \mathbf{C} then \mathbf{I}/\lesssim is a rational ICC.

Proof. (i) Since \lesssim is a CCC precongruence, it is also an ICC precongruence therefore the quotiented category \mathbf{I}/\lesssim is well-defined. Since \mathbf{I} is a subcategory of \mathbf{C} , each equivalent class of morphisms of \mathbf{I} is a subset of some equivalent class of morphisms of \mathbf{C} . Therefore, up to an obvious isomorphism, the category \mathbf{I}/\lesssim is a subcategory of \mathbf{C}/\lesssim . Finally, the incremental closure of \mathbf{I} immediately implies that of \mathbf{I}/\lesssim .

(ii) Suppose \mathbf{I} is a rational sub-ICC. By definition this means that \mathbf{C} is rational and \mathbf{I} is complete with respect to the operation \cdot^∇ . By Lemma 6.1.2, \mathbf{C}/\lesssim is also a rational CCC, therefore by (i), \mathbf{I}/\lesssim is a sub-ICC of a rational CCC.

Let $[f] : A \times B \rightarrow B$ be an equivalence class morphism in \mathbf{I}/\lesssim . It is also a morphism of the category \mathbf{C}/\lesssim , therefore by CCC rationality the least upper bound of the chain $[f]^{(n)}$ is given by $[f^\nabla]$ [AMJ94]. Since \mathbf{I} is \cdot^∇ -complete this implies that $[f^\nabla]$ is also in \mathbf{I}/\lesssim . Thus \mathbf{I}/\lesssim is also \cdot^∇ -complete.

Hence \mathbf{I}/\lesssim is a rational sub-ICC of \mathbf{C}/\lesssim . □

6.1.5 The internal language of incremental closed categories

By a well-known result by Lambek, the simply typed lambda calculus is the language of cartesian closed categories [Lam86]: for any cartesian closed category \mathbf{C} one can construct a typed lambda calculus $L(\mathbf{C})$ called the *internal language* of the CCC; and for any typed lambda calculus \mathcal{L} we can construct a CCC $Cl(\mathcal{L})$ that soundly interprets \mathcal{L} . This category is called the CCC *generated* by \mathcal{L} or also the *canonical classifying category* of \mathcal{L} [Cro93]. Furthermore these two transformations establish an equivalence of categories which means that their composites are naturally isomorphic to the identity functors:

$$\mathbf{C} \cong Cl(L(\mathbf{C})), \quad \mathcal{L} \cong L(Cl(\mathcal{L})) . \tag{6.1}$$

Does a similar correspondence hold between ICCs and safe typed lambda calculi? Following [Lam86], it is possible to adapt the notion of *internal language* to ICCs. Given an ICC \mathbf{I} , the formation rules of the internal language $L(\mathbf{I})$ of \mathbf{I} can be defined similarly to what Lambek did for CCC, but because the evaluation maps in an ICC are incremental and because an ICC is closed by incremental currying only, the abstraction and application rules are restricted accordingly: there is a side-condition imposing that the context variables have order greater than the order of the term being formed. The terms that can be formed in this language are precisely the *long-safe* terms of the internal language of the CCC:

$$L(\mathbf{I}) \stackrel{\text{def}}{=} \text{long-safe}(L(\mathbf{C}))$$

where $\text{long-safe}(\mathcal{L})$ denotes the long-safe fragment of a typed-lambda calculus \mathcal{L} .

Conversely, let \mathcal{L} be a typed-lambda calculus, we define the *canonical classifying category* of the long-safe fragment of \mathcal{L} , written $Cl(\text{long-safe}(\mathcal{L}))$, as the canonical sub ICCs of the canonical classifying category of \mathcal{L} :

$$Cl(\text{long-safe}(\mathcal{L})) \stackrel{\text{def}}{=} \text{subICC}(Cl(\mathcal{L})) .$$

Combining the last two equations with (6.1) gives for any ICC \mathbf{I} and long-safe language \mathcal{L} :

$$\mathbf{I} \cong Cl(L(\mathbf{I})), \quad \mathcal{L} \cong L(Cl(\mathcal{L})) . \quad (6.2)$$

Intrinsically safe fragment Let \mathbf{I} be an ICC. By definition it is isomorphic to a subcategory $\Pi(\mathbf{I})$ of some CCC \mathbf{C} for some injective functor $\Pi : \mathbf{I} \rightarrow \mathbf{C}$. We define the *intrinsically safe fragment* $LI(\mathbf{I})$ of $L(\mathbf{C})$ as the language consisting of the terms whose denotations in $\mathbf{C} \cong Cl(L(\mathbf{C}))$ are also in $\Pi(\mathbf{I})$:

$$LI(\mathbf{I}) \stackrel{\text{def}}{=} \{ t \in L(\mathbf{C}) \mid \llbracket t \rrbracket \in \text{Hom}(\Pi(\mathbf{I})) \} .$$

This definition implies $\llbracket LI(\mathbf{I}) \rrbracket = \Pi(\mathbf{I}) \cong I$. This language verifies the basic property of the safe lambda calculus:

Lemma 6.1.4. *Let \mathbf{I} be an ICC. For any term M of $LI(\mathbf{I})$, the free variables of M have order greater than $\text{ord } M$.*

Proof. Lambek [Lam86] defines a functor $\llbracket \cdot \rrbracket : \mathcal{L} \rightarrow \mathbf{C}$ such that every term M of the language \mathcal{L} of type B with free variables of type A_1, \dots, A_n is denoted by a morphism in $\mathbf{C}(A_1 \times \dots \times A_n, B)$. Take \mathcal{L} to be $LI(\mathbf{I})$, then by definition M is denoted by an incremental morphism therefore $\text{dro}(A_1 \times \dots \times A_n) \geq \text{ord } B$. We then have for $1 \leq i \leq n$:

$$\text{ord } A_i \geq \text{dro } A_i \geq \text{dro}(A_1 \times \dots \times A_n) \geq \text{ord } B . \quad \square$$

The language $LI(\mathbf{I})$, however, is *not* the safe fragment of the internal language of \mathbf{C} . Indeed, since safety is only preserved by β -reduction but not by β -equality, it is possible to have an unsafe term U in $L(\mathbf{C})$ with a safe beta-nf $\beta_{\text{nf}}(U)$. Since $\beta_{\text{nf}}(U)$ is safe, its denotation is an incremental morphism and therefore it belongs to $LI(\mathbf{I})$. But by soundness of the model \mathbf{C} , U and $\beta_{\text{nf}}(U)$ have the same denotation, therefore they both belong to $LI(\mathbf{I})$.

6.2 The game model

Our aim for the rest of this chapter is to construct a category of games that is incremental closed, thus giving rise to a game model of the safe lambda calculus. We start by introducing the class of *closed P-incremental justified strategies* and then show that it is closed under composition. This then allows us to construct an ICC category with game as objects and closed P-incremental justified strategies as morphisms.

For the rest of this chapter we make the following assumptions on games. Let \perp denote the game with a single initial move and no answers, for any game $A \neq \perp$:

(A1) Each question move in the game enables at least one answer move;

(A2) Answer moves do not enable any other move.

Clearly, PCF and IA games all verify these two assumptions. A game is said to be **prime** if it has a single initial move; a type is prime if its game denotation is prime.

6.2.1 Order of a move

Let $A = \langle M, \lambda, \vdash \rangle$ be a game. We call \vdash -chain, any sequence of enabling moves $m \vdash m_2 \vdash \dots \vdash m_h$ where $h \in \mathbb{N}$ is the *length* of the chain. The **order of a move** m in A , written $\text{ord}_A m$ (or just $\text{ord } m$ where there is no ambiguity) is defined as the length of the longest \vdash -chain starting from m minus 2. The **order of a game** is defined as the maximal order of its (initial) moves: $\text{ord } A = \max_{m \in M} \text{ord}_A m$. The **level** of a move m , written $\text{level}_A m$, is the length of the longest \vdash -chain ending with m . It is easy to see that the following relation holds:

$$\text{ord}_A m + \text{level}_A m \leq \text{ord } A .$$

We recall that for any type T built up from base types, product and function space, the order of T , written $\text{ord } T$, is defined by induction as follows: a base type has order 0, $\text{ord}(A \rightarrow B) = \max(1 + \text{ord } A, \text{ord } B)$, and $\text{ord}(A \times B) = \max(\text{ord } A, \text{ord } B)$ for any types A and B . Clearly, this definition coincides with the definition given above: the order of a type is the order of the arena denoting it: $\text{ord } T = \text{ord } \llbracket T \rrbracket$ for all type T .

Because of assumptions (A1) and (A2), for any move m of $A \neq \perp$, m is a question move if and only if $\text{ord } m \geq 0$, and m is an answer move if and only if $\text{ord } m = -1$.

Move-order after composition

Consider the game $X \multimap Y$ and let m be a move of $X \multimap Y$. We write $\text{ord}_{X \multimap Y} m$ to denote the order of m in the game $X \multimap Y$. If m belongs to X (resp. Y) then we write $\text{ord}_X m$ (resp. $\text{ord}_Y m$) to denote the order of the move m in the game X (resp. Y).

Lemma 6.2.1. *Let A , B and C be three games. We have:*

$$\begin{array}{ll} \forall m \in A : & \text{ord}_{A \multimap B} m = \text{ord}_{A \multimap C} m , \\ \forall m \in B : & \text{ord}_{A \multimap B} m \geq \text{ord}_{B \multimap C} m \quad \text{for } m \text{ initial,} \\ & \text{ord}_{A \multimap B} m = \text{ord}_{B \multimap C} m \quad \text{for } m \text{ non initial,} \\ \forall m \in C : & \text{ord}_{A \multimap C} m \geq \text{ord}_{B \multimap C} m \iff \text{ord } A \geq \text{ord } B \quad \text{for } m \text{ initial,} \\ & \text{ord}_{A \multimap C} m = \text{ord}_{B \multimap C} m \quad \text{for } m \text{ non initial.} \end{array}$$

The proof is immediate.

6.2.2 Well-bracketing

We call **pending question** of a sequence of moves $s \in L_A$ the last unanswered question in s .

Definition 6.2.1. A strategy σ is said to be **P-well-bracketed** if for any play $sa \in \sigma$ where a is a P-answer, a points to the pending question in s .

P-well-bracketing can be restated differently as the following proposition shows:

Proposition 6.2.1. *We make assumption (A1) and (A2). Let σ be a strategy on a game A . The following statements are equivalent:*

- (i) σ is P-well-bracketed,
- (ii) for $sa \in \sigma$ with a a P-answer, a points to the pending question in $\ulcorner s \urcorner$,
- (iii) for $sa \in \sigma$ with a a P-answer, a points to the last O-question in $\ulcorner s \urcorner$,

(iv) for $s a \in \sigma$ with a a P-answer, a points to the last O-move in $\lceil s \rceil$ with order $> \text{ord } a$.

Proof. The result holds trivially if $A = \perp$ (the game with one initial question and no answers). Otherwise:

(i) \iff (ii): [McC96a, Lemma 2.1] states that if P is to move then the pending question in s is the same as that of $\lceil s \rceil$.

(ii) \iff (iii): Assumption (A2) implies that the pending question in $\lceil s \rceil$ is also the last O-question occurring in $\lceil s \rceil$.

(iii) \iff (iv): Because of assumption (A1) and (A2), for any move m , we have m is a question move if and only if $\text{ord } m \geq 0$ if and only if $\text{ord } m > \text{ord } a = -1$. \square

Lemma 6.2.2. *Under assumption (A2), if s be a justified sequence of moves satisfying alternation and visibility then any O-move (resp. P-move) in s points to an unanswered P question (resp. O-question).*

Proof. Suppose that an O-move c points to a P-move d that has already been answered by the O-move a . The sequence s as the following form:

$$s = \dots d \dots a \dots c .$$

By O-visibility, d must belong to $\downarrow s_{<c}$. But since a is an answer, by assumption (A2), it cannot justify any P-move, therefore $\downarrow s_{<q}$ must contain an OP-arc “hoping” over a . We name the nodes of this arc d^1 and c^1 :

$$s = \dots d \dots d^1 \dots a \dots c^1 \dots c .$$

By P-visibility, d^1 must belong to $\lceil s_{<c^1} \rceil$. Consequently, a does not belong to $\lceil s_{<c^1} \rceil$ (otherwise the PO-arc $\overrightarrow{d^1 a}$ would cause the P-view to jump over d^1). Therefore there must be a PO-arc $\overrightarrow{d^2 c^2}$ in $\lceil s_{<c^1} \rceil$ hoping over a :

$$s = \dots d \dots d^1 \dots c^2 \dots a \dots d^2 \dots c^1 \dots c$$

This process can be repeated infinitely often by using alternatively O-visibility and P-visibility. This gives a contradiction since the sequence of moves $s_{<c}$ has finite length. Hence d cannot point to a question that has already been answered. Since, by assumption (A2), a question is enabled by another question, d is necessarily justified by an unanswered question. \square

Lemma 6.2.3. *Under assumption (A2), if s is a P-well-bracketed justified sequence of moves of odd length satisfying alternation and visibility then all O-questions occurring in $\lceil s \rceil$ are unanswered in s .*

Proof. We proof the first part by induction on s . The base case ($s = q$ with q initial O-move) is trivial. Suppose $s = s' \cdot q \cdot u \cdot m$. We have $\lceil s \rceil = \lceil s' \rceil \cdot q \cdot m$. Clearly m is unanswered in s . Let r be an O-question in $\lceil s' \rceil$ and suppose that r is answered in s by some move a . By the induction hypothesis, r is unanswered in s' therefore a necessarily appears in the segment u :

$$s = \underbrace{\dots r^O \dots}_{s'} q^P \underbrace{\dots a^P \dots}_u m^O .$$

But since m is justified by q , by Lemma 6.2.2 q must be unanswered in $s_{<m}$. In particular, the pending question at $s_{\leq a}$ cannot be r since the unanswered question q is played after r . This gives a contradiction since by well-bracketing a should answer the pending question. Hence r is unanswered in s . \square

6.2.3 P-incremental justification

P-incremental justification is a generalization of well-bracketing to question moves:

Definition 6.2.2. A play sm of even length is said to be **P-incrementally justified**, or **P-i.j.** for short, if m points to the last unanswered O-question in $\lceil s \rceil$ with order strictly greater than $\text{ord } m$. A strategy σ is said to be **P-incrementally justified**, if all plays in σ ending with a P-question are P-incrementally justified.

Let σ be a strategy. We write $\mathcal{P}(\sigma)$ to denote the subset of σ consisting of plays whose even-length prefixes are all P-i.j. Hence P-i.j. strategies are precisely those verifying the relation $\sigma = \mathcal{P}(\sigma)$.

Proposition 6.2.2. *Let σ be a P-well-bracketed strategy on a game A . Under assumptions (A1) and (A2), the following statements are equivalent:*

- (i) σ is P-incrementally justified,
- (ii) for $sq \in \sigma$ with q a P-question, q points to the last O-question in $\lceil s \rceil$ with order $> \text{ord } q$,
- (iii) for $sq \in \sigma$ with q a P-question, q points to the last O-move in $\lceil s \rceil$ with order $> \text{ord } q$.

Proof. The result holds trivially if $A = \perp$ (the game with one initial question and no answers). Otherwise: (i) iff (ii): By Lemma 6.2.3, O-questions occurring in $\lceil s \rceil$ are all unanswered. (ii) iff (iii): By (A1), $\text{ord } q \geq 0$ and by (A2), answer moves have order 0 therefore answer moves all have order $\leq \text{ord } q$. \square

Putting Proposition 6.2.2 and 6.2.1 together we obtain:

Proposition 6.2.3. *Under assumption (A1) and (A2), a strategy σ is P-well-bracketed and P-incrementally justified if and only if for $sm \in \sigma$, m points to the last O-move in $\lceil s \rceil$ with order $> \text{ord } m$.*

6.2.4 Closed P-incremental justification

Definition 6.2.3. Let sm be an even-length play on some game $A \rightarrow B$. sm is said to be **closed** **P-incrementally justified** (closed P-i.j. for short) just if

- (i) sm is P-incrementally justified;
- (ii) and if m is an initial move in A then its justifier n (initial in B) verifies $\text{ord}_A m \geq \text{ord}_B n$.

A strategy σ is **closed P-i.j.** just if all plays in σ ending with a P-questions are closed P-i.j.

Example 6.2.1. For any game A , the identity strategy id_A is closed P-i.j.

Lemma 6.2.4. *Let $\sigma : A \multimap B$ be a P-i.j. strategy.*

- (i) *If for each initial move m of A occurring in some play of σ we have $\text{ord}_A m \geq \text{ord } B$, then σ is closed P-i.j.*
- (ii) *Suppose that $A = A_1 \times \dots \times A_n$ where each of the A_i are prime arenas. If for each initial move m_i of A_i , for $i \in \{1..n\}$, occurring in some play of σ we have $\text{ord } A_i \geq \text{ord } B$, then σ is closed P-i.j.*

Proof. (i) This is a direct consequence of the definition since $\text{ord } B \geq \text{ord}_B b$ for every move b initial in B . (ii) Take an initial move m of A . We have $\text{ord}_A m = \text{ord}_{A_i} m$ for some i . This is in turn equal to $\text{ord } A_i$ since A_i is prime. By hypothesis it is greater than $\text{ord } B$ hence we can conclude using (i). \square

Example 6.2.2. The simply typed term $x : (o^1 \rightarrow o^2) \times o^3 \vdash \lambda y^o. \pi_2 x : o^4 \rightarrow o^5$ has a P-i.j. denotation. The second part of the previous Lemma cannot be used because its hypothesis is not verified, and indeed the denotation is not closed P-i.j. since it contains the play $q^5 q^3$ and we have $\text{ord}_{(o^1 \rightarrow o^2) \times o^3} q^3 = 0 < 1 = \text{ord}_{o^4 \rightarrow o^5} q^5$.

Observe that the “P-incremental justification” property is preserved across the *curry* isomorphism, but this is not the case for closed P-incremental justification. It is possible to have two isomorphic strategies σ and μ such that one is closed P-i.j. but not the other. For instance any strategy σ that is P-i.j. on the game $I \multimap A$ is also closed P-i.j. When seen as a strategy on the isomorphic game A , however, σ is not necessarily closed P-i.j.¹; thus the distinction between the games $I \multimap A$ and A matters. This is because the definition of closed P-i.j. strategy specifically refers to the moves of the arena in the left-hand side of the function space arrow \multimap . A consequence of this is that the category of closed P-i.j. strategies that we will introduce later on, is neither monoidal closed nor cartesian closed.

6.2.5 Interaction sequences

In this section we recall some basic definitions and results used in game semantics. We fix here some notations that will be used to analyze interaction sequences.

Let A, B and C be three games. We say that u is an **interaction sequence** of A, B and C whenever $u \upharpoonright A, B$ is a valid position of the game $A \multimap B$ (i.e., $u \upharpoonright A, B \in P_{A \multimap B}$) and $u \upharpoonright B, C$ is a valid position of the game $B \multimap C$. We write $\text{Int}(A, B, C)$ to denote the set of all such interaction sequences.

Let $\sigma : A \multimap B$ and $\mu : B \multimap C$ be two strategies. We write $\sigma \parallel \mu$ to denote the set of interaction sequences that unfold according to the strategy σ in the A, B -projection of the game and to μ in the B, C -projection:

$$\sigma \parallel \mu = \{u \in \text{Int}(A, B, C) \mid u \upharpoonright A, B \in \sigma \wedge u \upharpoonright B, C \in \mu\}.$$

The composite of σ and μ is then defined as $\sigma; \mu = \{u \upharpoonright A, C \mid u \in \sigma \parallel \mu\}$.

The diagram below shows the structure of an interaction sequence from $\sigma \parallel \mu$. There are four states represented by the rectangular boxes. The content of the state shows who is to play in each of the game $A \multimap B$, $B \multimap C$ and $A \multimap C$. For instance in state *OPP*, it is O’s turn to play in $A \multimap B$ and P’s turn to play in $B \multimap C$ and $A \multimap C$. Arrows represent the moves. When specifying interaction sequence, the following bullet symbols are used to represent moves: \circ for P-moves, \bullet for O-moves, \circlearrowleft for a move playing the role of P in $A \multimap B$ and O in $B \multimap C$ and \circlearrowright for the symmetric of \circlearrowleft . We sometimes add a subscript to the symbols \circ and \bullet to denote the component in which the moves is played (A or C).

Note that in state *OPP*, the alternation condition (for each of the three games involved) prevents the players from playing in A . Indeed, the O-moves in component A of $A \multimap B$ are also O-moves in component A of $A \multimap C$ however the state name indicates that the next move in $A \multimap B$ must be an O-move and the next move in $A \multimap C$ must be a P-move.

Similarly, in the top state *OOO*, the players cannot make move in B since the O-moves in component B of the game $B \multimap C$ correspond to P-moves in the component B of $A \multimap B$. However the state name indicates that the next move in $A \multimap B$ and the next move in $B \multimap C$ must be played by O.

Let $u \in \text{Int}(A, B, C)$ and m be a move of u . The **component** of m is A, B if after playing m the game is under the control of the strategy σ and B, C otherwise (if μ has control). In other words, the moves $\bullet, \circ \in A$ and $\circlearrowleft \in B$ shown on the diagram of Fig. 6.1 have component A, B and $\bullet, \circ \in C$ and $\circlearrowright \in B$ have component B, C .

Also we call **generalized O-move in component A, B** moves that play the role of O in the game $A \multimap B$, that is to say moves represented by \bullet and \bullet_A . Similarly \circlearrowright -moves and \circ_A -moves

¹In particular, every P-i.j. strategy σ on the game $!A_1 \otimes \dots \otimes !A_n \multimap B$, is isomorphic, up to arena-tagging of the moves, to the closed P-i.j. strategy $\Lambda^n(\sigma)$ on the game $I \multimap (A_1, \dots, A_n, B)$, where Λ denotes the *curry* isomorphism.

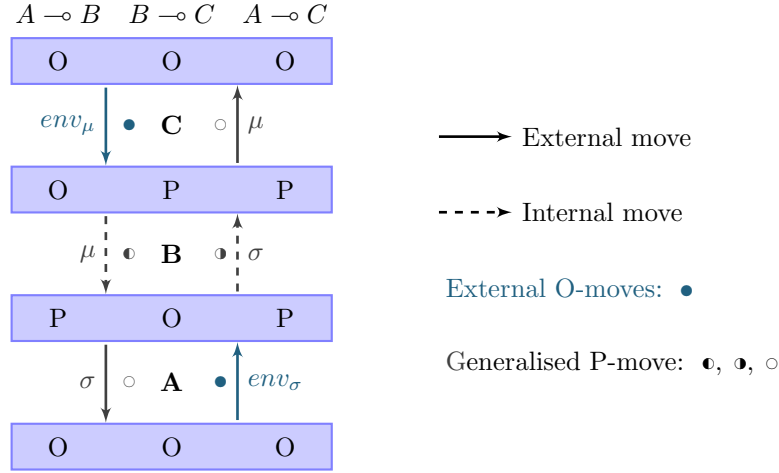


Figure 6.1: Structure of an interaction sequence.

are the **generalized P-moves in component A**, \bullet_C -moves and \circ -moves are the **generalized O-moves in component B, C** and \circ_C -moves and \bullet -moves are the **generalized P-moves in component B, C**.

The **P-view** of an interaction sequence $u \in \text{Int}(A, B, C)$ (also called *core* [McC96b]), written \bar{u} or $\lceil u \rceil$, is defined as:

$$\begin{aligned} \lceil u \cdot n \rceil &= n & \text{if } m \text{ is an external O-move initial in } C, \\ \lceil u \cdot m \cdot v \cdot n \rceil &= n & \text{if } m \text{ is an external O-move non initial in } C, \\ \lceil u \cdot m \rceil &= \lceil u \rceil \cdot m & \text{if } m \text{ is a generalised P-move.} \end{aligned}$$

Lemma 6.2.5. *Let u be an interaction sequence in $\text{Int}(A, B, C)$ then*

$$\lceil u \rceil \upharpoonright A, C = \lceil u \rceil \upharpoonright A, C^\top.$$

Proof. By induction on u . It is trivial for the empty sequence. Let b be a move in B . We have $\lceil u \cdot b \rceil \upharpoonright A, C = \lceil u \rceil \upharpoonright A, C$. By the I.H. this equals $\lceil u \upharpoonright A, C^\top \rceil = \lceil u \cdot b \upharpoonright A, C^\top \rceil$. Let m be a P-move in A or C then $\lceil u \cdot m \rceil \upharpoonright A, C = (\lceil u \rceil \upharpoonright A, C) \cdot m$ and by the I.H. it equals $\lceil u \upharpoonright A, C^\top \rceil \cdot m = \lceil (u \upharpoonright A, C) \cdot m \rceil = \lceil u \cdot m \upharpoonright A, C^\top \rceil$. Let c be an initial move in C . We have $\lceil u \cdot c \upharpoonright A, C^\top \rceil = \lceil (u \upharpoonright A, C) \cdot c \rceil = c = c \upharpoonright A, C = \lceil u \cdot c \upharpoonright A, C \rceil$. Let $u = u_1 \cdot \overbrace{m \cdot u_2 \cdot n}^{\text{O-move in } A \rightarrow C}$ with n an O-move in $A \rightarrow C$. Then necessarily $m \in A, C$ and $\lceil u \upharpoonright A, C^\top \rceil = \lceil u_1 \upharpoonright A, C \rceil \cdot \overbrace{m \cdot u_2 \upharpoonright A, C \cdot n}^{\text{O-move in } A \rightarrow C} = \lceil u_1 \upharpoonright A, C^\top \rceil \cdot \overbrace{m \cdot n}^{\text{O-move in } A \rightarrow C}$. By the I.H. it equals $(\lceil u_1 \upharpoonright A, C \rceil \cdot \overbrace{m \cdot n}^{\text{O-move in } A \rightarrow C}) \upharpoonright A, C = \lceil u_1 \cdot \overbrace{m \cdot n}^{\text{O-move in } A \rightarrow C} \upharpoonright A, C$ \square

We will also make use of another result that was used by Harmer to show compositionality of P-visible strategies [Har05]:

Lemma 6.2.6. [Har05, Lemma 3.3.1] *If $u \in \text{Int}(A, B, C)$ such that $u \upharpoonright A, B \in \sigma$ and $u \upharpoonright B, C \in \tau$ where σ, τ are two (P-visible) strategies, and m is a generalized O-move of u in component X then $\lceil u_{\leq m} \upharpoonright X \rceil = \lceil \bar{u}_{\leq m} \upharpoonright X \rceil$.*

NOTATIONS 6.2.1 We now introduce some notations for moves that will come useful when representing plays. The symbol \bullet stands for an O-move and \circ for a P-move. If the game considered is of the form $L \multimap R$ then we write \bullet_L and \circ_L (resp. \bullet_R and \circ_R) to represent a move that belongs to the component L (resp. R). For interaction sequences in $\text{Int}(A, B, C)$ we use the set of symbols $\{\bullet_A, \circ_A, \bullet_C, \circ_C, \bullet, \circ\}$ as defined in Fig. 6.1. We also identify each of these symbols

with the set of moves of the corresponding kind. Thus we write “ $m \in \bullet_A$ ” to mean that m is an O-move played in A . We use the variable X to denote either the component A, B or B, C , and the variable Y to denote the opposite component.

For any given component X , we write \circ_X to denote a generalized P-move in X and \bullet_X to denote a generalized O-move in X . Thus $\bullet_{A,B} = \bullet$, $\circ_{A,B} = \circ$, $\bullet_{B,C} = \bullet$, and $\circ_{B,C} = \circ$. We write \bullet_X (resp. \circ_X) to denote an external O-move (resp. P-move) in component X . Thus $\bullet_{A,B} = \bullet_A$, $\circ_{A,B} = \circ_A$, $\bullet_{B,C} = \bullet_C$, and $\circ_{B,C} = \circ_C$. We write $s \sqsubseteq t$ to say that s is a subsequence (with pointers) of t , $s \leq t$ to say that s is a prefix (with pointers) of t and $s \geq t$ to say that s is a suffix of t .

6.2.6 Preliminary results

In this section, we prove several preliminary lemmas which will help us to study compositionality of P-i.j. strategies.

Lemma 6.2.7. *Let X be a component (either A, B or B, C). Let u be an interaction sequence of the form $u = \dots \beta \dots n \dots \alpha \dots m$ where:*

$$\begin{array}{ccc} & \curvearrowright & \\ \circ_X & & \bullet_X \end{array}$$

- α, β are external moves in component X (necessarily both played in A or in C),
- m is either played in B or an external P-move in X ,
- α is visible at m in X (i.e., $\alpha \in \ulcorner u \upharpoonright X \urcorner$) and consequently β is also visible.

Then $n \notin \ulcorner u \upharpoonright A, C \urcorner$.

Proof. Since α is an O-move, α and β are necessarily played in the same arena (A or C). Take $v = u$ if m is a generalized O-move in X and $v = u_{<z}$ otherwise (if m is a generalized P-move in X). The third assumption implies $\alpha, \beta \in \ulcorner v \urcorner$. The last move in v is necessarily a generalized O-move in component X (see diagram of Fig. 6.1) therefore by Lemma 6.2.6 we have $\ulcorner v \upharpoonright X \urcorner = \ulcorner \bar{v} \upharpoonright X \urcorner \sqsubseteq \bar{v} \sqsubseteq \bar{u}$. Thus $\alpha, \beta \in \bar{u}$ and since α, β are played in A, C we have $\alpha, \beta \in \bar{u} \upharpoonright A, C = \ulcorner u \upharpoonright A, C \urcorner$ (Lemma 6.2.5). Finally since n lies underneath the β - α PO-arc it cannot appear in the P-view $\ulcorner u \upharpoonright A, C \urcorner$. \square

Lemma 6.2.8. *Let u be an interaction sequence in $\text{Int}(A, B, C)$ and n be a move of u such that $n \in \ulcorner u \upharpoonright A, C \urcorner$:*

- i. *if all the moves in $u_{\geq n}$ are played in C then $n \in \ulcorner u \upharpoonright B, C \urcorner$;*
- ii. *if all the moves in $u_{\geq n}$ are played in A then $n \in \ulcorner u \upharpoonright A, B \urcorner$.*

Proof. (i) We show the contrapositive. Suppose that $n \notin \ulcorner u \upharpoonright B, C \urcorner$ then either:

- $\ulcorner u \upharpoonright B, C \urcorner$ contains an initial move $c_0 \in C$ occurring after n in u .

By Lemma 6.2.6 we have $\ulcorner u \upharpoonright B, C \urcorner = \ulcorner \bar{u} \upharpoonright B, C \urcorner \sqsubseteq \ulcorner u \urcorner$, thus c_0 also occurs in $\ulcorner u \urcorner$. Since c_0 belongs to C we have $c_0 \in \ulcorner u \urcorner \upharpoonright A, C = \ulcorner u \upharpoonright A, C \urcorner$ (Lemma 6.2.5). Thus the P-view $\ulcorner u \upharpoonright A, C \urcorner$ starts with the initial move c_0 and since n occurs before c_0 , n does not occur in the P-view.

- or n lies underneath a PO-arc β - α visible at $u \upharpoonright B, C$. By assumption, since α occurs after n in u , it must belong to C . We can therefore apply Lemma 6.2.7 with $X \leftarrow B, C$ which gives $n \notin \ulcorner u \upharpoonright A, C \urcorner$.

(ii) Suppose that $n \notin \ulcorner u \upharpoonright A, B \urcorner$ then either:

- $\ulcorner u \upharpoonright A, B \urcorner$ contains an initial move $b_0 \in B$ occurring after n in u . But this is impossible since by assumption all the moves occurring after n in u belong to A .

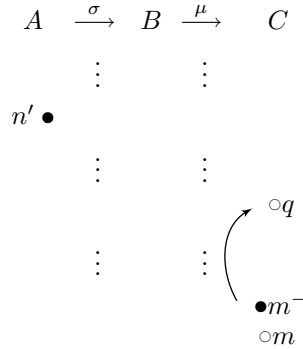
- or n lies underneath a PO-arc $\beta\text{-}\alpha$ in A, B . By assumption, since α occurs after n it must belong to A . We can then conclude using Lemma 6.2.7 with $X \leftarrow A, B$. \square

Note that we cannot completely relax the assumption which says that moves in $u_{\geq n}$ are all in the same component. For instance take $u = \bullet_C \overset{n}{\curvearrowright} \bullet_A \bullet$ then we have $n \in \ulcorner u \urcorner \upharpoonright A, C^\top$ but $n \notin \ulcorner u \urcorner \upharpoonright A, B^\top$.

Lemma 6.2.9 (P-visibility decomposition (from C)). *Let $u = \dots n' \cdot r \cdot m \in \text{Int}(A, B, C)$ where n' is a \bullet_A -move verifying $n' \in \ulcorner u \urcorner \upharpoonright A, C^\top$ and m is in $\circ_C \cup \bullet \cup \bullet$. Then there is a \bullet -move γ in $r \cdot m$ such that $\gamma \in \ulcorner u \urcorner \upharpoonright B, C^\top$, $n' \in \ulcorner u_{\leq \gamma} \urcorner \upharpoonright A, B^\top$ and γ is justified by a move occurring before n' .*

Proof. By induction on $|r|$. If $r = \epsilon$ then necessarily $u = \dots \bullet_A \bullet$ where m points before n' (n' being played in A cannot justify m played in B) so we just need to take $\gamma = m$. If $|r| = 1$ then either $u = \dots \bullet_A \bullet \circ_C$ or $u = \dots \bullet_A \bullet \bullet$. In both cases we can take γ to be the \bullet -move between n' and m . Suppose $|r| > 1$. Let m^- denote the move preceding m in u . We proceed by case analysis:

- Suppose $m \in \circ_C$ and $m^- \in \bullet_C$. Let q be the external P-move that justifies m^- . Since $n' \in \ulcorner u \urcorner \upharpoonright A, C^\top$, q must occur after n' in u :




Thus we can use the induction hypothesis (with $u \leftarrow u_{\leq q}$): there is a \bullet -move γ in $u_{[n', q]}$ pointing before n' such that $\gamma \in \ulcorner u_{\leq q} \urcorner \upharpoonright B, C^\top$, $n' \in \ulcorner u_{\leq \gamma} \urcorner \upharpoonright A, B^\top$. Moreover $\ulcorner u_{\leq q} \urcorner \upharpoonright B, C^\top \leq \ulcorner u_{\leq m} \urcorner \upharpoonright B, C^\top$ (since q is visible from m in B, C) thus we have $\gamma \in \ulcorner u_{\leq m} \urcorner \upharpoonright B, C^\top$ as required.

- Suppose $m \in \circ_C$ and $m^- \in \bullet$. Again we can conclude using the induction hypothesis with $u \leftarrow u_{\leq m^-}$.
- Suppose $m \in \bullet$.

Suppose that all the moves in r are in A . Then r is of the form $(\circ_A \bullet_A)^*$ (where $(\cdot)^*$ denotes the Kleene star operator). We just need to take $\gamma = m$. Indeed, moves in $u_{\geq m}$ are all in A and by assumption $n' \in \ulcorner u \urcorner \upharpoonright A, C^\top$ therefore Lemma 6.2.8(ii) gives $n' \in \ulcorner u \urcorner \upharpoonright A, B^\top$. Also, since m is a \bullet -move, its justifier is a \bullet -move but r contains only \bullet and \circ moves hence m 's justifier must occur before n' .

Suppose that r contains at least one move in B . Let b be the last such move, then u is of the form $\dots n' \dots \bullet \cdot (\circ_A \bullet_A)^* \cdot \bullet$. We then have $u \upharpoonright B, C = \dots n' \dots \bullet \cdot \bullet$ thus $b \in \ulcorner u \urcorner \upharpoonright B, C^\top$.

We can then conclude by applying the induction hypothesis with $u \leftarrow u_{\leq b}$.

- Suppose m  If $m^- \in \bullet$ then the I.H. with $u \leftarrow u_{\leq m^-}$ permits us to conclude. If $m^- \in \bullet_C$ then we conclude by applying the I.H. on $u \leftarrow u_{\leq q}$ where q is the external P-move in C justifying m^- . \square

We now show the symmetric of the previous lemma:

Lemma 6.2.10 (P-visibility decomposition (from A)). *Let $u = \dots n' \cdot r \cdot m \in \text{Int}(A, B, C)$ where n' is an O -move non initial in C verifying $n' \in \ulcorner u \upharpoonright A, C^\top$ and m is in $\circ_A \cup \bullet \cup \circ$. Then there is a \bullet -move γ in $r \cdot m$ such that $\gamma \in \ulcorner u \upharpoonright A, B^\top$, $n' \in \ulcorner u_{\leq \gamma} \upharpoonright B, C^\top$ and γ is justified by a move occurring before n' .*

Proof. The proof is almost symmetrical to the previous one (Lemma 6.2.9). We proceed by induction on $|r|$. If $r = \epsilon$ then necessarily $u = \dots \bullet_C \bullet$ where m points before n' (it cannot point to n' since n' is not initial in C). Thus we just need to take $\gamma = m$.

If $|r| = 1$ then either $u = \dots \bullet_C \bullet \circ_A$ or $u = \dots \bullet_C \bullet \circ$. In both cases we can take γ to be the \bullet -move between n' and m . Suppose $|r| > 1$. Let m^- denote the move preceding m in u . We do a case analysis:

- Suppose $m \in \circ_A$ and $m^- \in \bullet_A$. Let q be the external P-move that justifies m^- . Since $n' \in \ulcorner u \upharpoonright A, C^\top$, q must occur after n' in u :

$$\begin{array}{ccccc}
 A & \xrightarrow{\sigma} & B & \xrightarrow{\mu} & C \\
 & & \vdots & & \vdots \\
 & & & & \bullet n' \\
 & & \vdots & & \vdots \\
 q \circ & & \vdots & & \vdots \\
 \uparrow & & \vdots & & \vdots \\
 m^- \bullet & & & & \\
 m \circ & & & &
 \end{array}$$

Thus we can use the induction hypothesis (with $u \leftarrow u_{\leq q}$): there is a \bullet -move γ in $u_{[n', q]}$ pointing before n' such that $\gamma \in \ulcorner u_{\leq q} \upharpoonright A, B^\top$, $n' \in \ulcorner u_{\leq \gamma} \upharpoonright B, C^\top$. Moreover $\ulcorner u_{\leq q} \upharpoonright A, B^\top \leq \ulcorner u_{\leq m} \upharpoonright A, B^\top$ (since q is visible from m in A, B) thus we have $\gamma \in \ulcorner u_{\leq m} \upharpoonright A, B^\top$ as required.

- Suppose $m \in \circ_A$ and $m^- \in \circ$ then again we can conclude using the I.H. with $u \leftarrow u_{\leq m^-}$.

- Suppose $m \in \bullet$.

- Suppose that r does not contain any move in B then r is of the form $(\circ_C \bullet_C)^*$.

We just need  take $\gamma = m$. Indeed:

- By lemma 6.2.8(i) we have $n' \in \ulcorner u \upharpoonright B, C^\top$.
- m is justified by a move occurring before n' . Indeed, if m is justified by a \bullet -move then since $n' \cdot r$ contains only \bullet and \circ moves, m 's justifier must occur before n' . If m 's justifier is an initial \bullet_C -move c_i , then by P-visibility we have $c_i \in \ulcorner u \upharpoonright B, C^\top$ but since the P-view computation “stops” when reaching an initial moves, and because by (a) n' also belongs to the P-view, n ; necessarily occurs after c_i .

- Suppose that r contains some move in B . Let b be the last such move. Then u is of the form $u = \dots n' \cdot \dots \bullet \cdot (\circ_A \bullet_A)^* \cdot \bullet$. So we have $u \upharpoonright B, C = \dots n' \cdot \dots \bullet \cdot \bullet$ hence $b \in \ulcorner u \upharpoonright B, C^\top$.

We can now conclude by applying the I.H. with $u \leftarrow u_{\leq b}$.

- iv. Suppose $m \in \bullet$. If $m^- \in \bullet$ then the I.H. with $u \leftarrow u_{\leq m^-}$ permits us to conclude. If $m^- \in \bullet_A$ then we conclude by applying the I.H. on $u \leftarrow u_{\leq q}$ where q is the external P-move in A justifying m^- . \square

Using the two preceding Lemmas we can show:

Lemma 6.2.11 (Increasing order lemma) *Let $u = \dots n' \cdot r \cdot m \in \text{Int}(A, B, C)$ where*

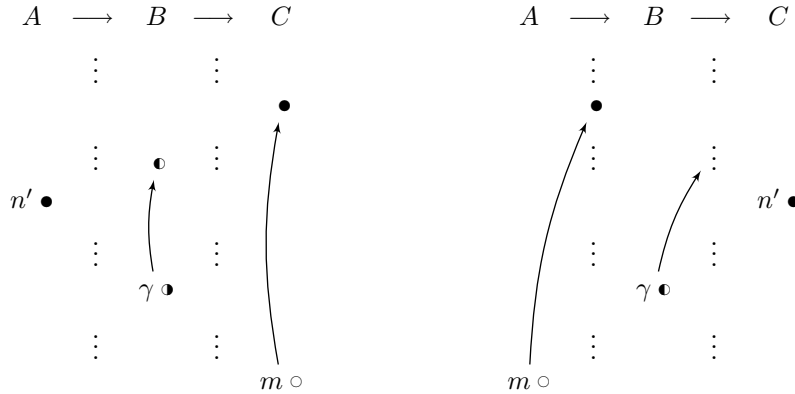
1. n' is an external O-move in component X ($n' \in \bullet_A$ and $X = A, B$, or $n' \in \bullet_C$ and $X = B, C$) non initial in C ,
2. $n' \in \ulcorner u \upharpoonright A, C \urcorner$,
3. m is either played in B (in \bullet or \bullet) or is an external P-move in Y (in \circ_C if $n' \in \bullet_A$, or in \circ_A if $n' \in \bullet_C$),
4. m 's justifier occurs before n' ,
5. $u \upharpoonright Y$ is P-i.j.,
6. $u_{\leq b} \upharpoonright X$ is P-i.j. for every B-move b occurring in u such that b is a generalized P-move in X and is not initial in B .

Then:

$$\text{ord}_Y m \geq \text{ord}_{A \rightarrow C} n'.$$

Proof. If $n' \in \bullet_C$ (resp. if $n' \in \bullet_A$) then by Lemma 6.2.10 (resp. Lemma 6.2.9) there is an occurrence in $r \cdot m$ of a non-initial B-move γ of type \bullet (resp. \bullet) such that $\gamma \in \ulcorner u \upharpoonright Y \urcorner$, $n' \in \ulcorner u_{\leq \gamma} \upharpoonright X \urcorner$ and γ is justified by a move occurring before n' .

There are six possible cases depending on the type of the moves n' and m : $(n', m) \in \bullet_A \times (\circ_C \cup \bullet \cup \bullet) \cup \bullet_C \times (\circ_A \cup \bullet \cup \bullet)$. The following diagram illustrates the cases $(n', m) \in \bullet_A \times \circ_C$ (left) and $(n', m) \in \bullet_C \times \circ_A$ (right):



We have:

$$\text{ord}_Y \gamma \geq \text{ord}_X \gamma \tag{6.3}$$

Indeed, if $n' \in \bullet_C$ then $X = B, C$ and $Y = A, B$ and by Lemma 6.2.1 we have $\text{ord}_{A \rightarrow B} \gamma \geq \text{ord}_{B \rightarrow C} \gamma$. If $n' \in \bullet_A$ then γ is a \bullet -move therefore it is not initial in B and Lemma 6.2.1 gives $\text{ord}_{A \rightarrow B} \gamma = \text{ord}_{B \rightarrow C} \gamma$.

Hence:

$$\begin{aligned} \text{ord}_{A \rightarrow C} n' &= \text{ord}_X n' && (\text{n' non initial in } C \text{ \& Lemma 6.2.1}) \\ &\leq \text{ord}_X \gamma && (u_{\leq \gamma} \upharpoonright X \text{ is P-i.j. by the 6}^{th} \text{ hyp. \& } \gamma\text{'s justifier occurs before } n') \end{aligned}$$

$$\begin{aligned}
&\leq \text{ord}_Y \gamma && \text{(By Eq. 6.3)} \\
&\leq \text{ord}_Y m && (u \upharpoonright Y \text{ is P-i.j. \& by 4}^{th} \text{ hyp. } m\text{'s justifier occurs before } \gamma) \quad \square
\end{aligned}$$

Lemma 6.2.12. *Let $u \in \text{Int}(A, B, C)$ such that $u = \dots \gamma \dots \delta \dots m$ where m is a generalized P-move in X , $\gamma \in \ulcorner u \upharpoonright A, C \urcorner$ and $\delta \in \ulcorner u \upharpoonright X \urcorner$. Then $\gamma \in \ulcorner u_{\leq \delta} \upharpoonright A, C \urcorner$.*

Proof. First we remark that δ must occur in $\ulcorner u \urcorner$. Indeed, $\delta \in \ulcorner u \upharpoonright X \urcorner = \ulcorner u_{< m} \upharpoonright X \urcorner \cdot m$ therefore $\delta \in \ulcorner u_{< m} \upharpoonright X \urcorner$ and since the move preceding m in u is necessarily a generalized O-move in X , we can apply Lemma 6.2.6:

$$\begin{aligned}
\delta \in \ulcorner u_{< m} \upharpoonright X \urcorner &= \ulcorner \ulcorner u_{< m} \urcorner \upharpoonright X \urcorner && \text{by Lemma 6.2.6} \\
&\sqsubseteq \ulcorner u_{< m} \urcorner \\
&\sqsubseteq \ulcorner u \urcorner .
\end{aligned}$$

Clearly, $\ulcorner u_{\leq \delta} \upharpoonright A, C \urcorner$ is a prefix of $\ulcorner u \upharpoonright A, C \urcorner$, indeed:

$$\begin{aligned}
\ulcorner u_{\leq \delta} \upharpoonright A, C \urcorner &= \ulcorner u_{\leq \delta} \urcorner \upharpoonright A, C && \text{(Lemma 6.2.5)} \\
&\leq \ulcorner u \urcorner \upharpoonright A, C && (\delta \in \ulcorner u \urcorner) \\
&= \ulcorner u \upharpoonright A, C \urcorner && \text{(Lemma 6.2.5) .}
\end{aligned}$$

Finally since $\gamma \in \ulcorner u \upharpoonright A, C \urcorner$ and γ occurs before δ in u , we necessarily have $\gamma \in \ulcorner u_{\leq \delta} \upharpoonright A, C \urcorner$. \square

Lemma 6.2.13. *Let X be a component and $u \in \text{Int}(A, B, C)$ such that the projection of u on the component X has the form:*

$$u \upharpoonright X = \dots n \dots n' \dots m$$

$\bullet_X \quad \circ_X$

and

1. m and n' are external move in X (in A if $X = A, B$ and in C if $X = B, C$);
2. $u \upharpoonright X$ is P-i.j.;
3. $u_{\leq b} \upharpoonright A, B$ is P-i.j. for every \bullet -move b occurring in u ;
4. $u_{\leq b} \upharpoonright B, C$ is P-i.j. for every \bullet -move b not initial in B occurring in u .

Then either $\text{ord}_{A \rightarrow C} n' \leq \text{ord}_{A \rightarrow C} m$ or $n' \notin \ulcorner u \upharpoonright A, C \urcorner$.

Proof. - Suppose that n' occurs in the P-view $\ulcorner u \upharpoonright X \urcorner$. Then we have

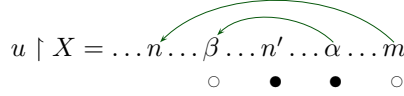
$$\text{ord}_{A \rightarrow C} n' = \text{ord}_{B \rightarrow C} n' . \tag{6.4}$$

Indeed, if X is the component B, C then necessarily n' is not initial in C (otherwise it would be the first move in $\ulcorner u \upharpoonright B, C \urcorner$, which is not the case since by visibility n must occur before n' in the P-view) and if $X = A, B$ then n' is in A . Thus in both cases, Lemma 6.2.1 gives us the claimed equality.

Thus,

$$\begin{aligned}
\text{ord}_{A \rightarrow C} n' &= \text{ord}_X n' && \text{(Eq. 6.4)} \\
&\leq \text{ord}_X m && (u \upharpoonright X \text{ is P-i.j.}) \\
&= \text{ord}_{A \rightarrow C} m && \text{(Lemma 6.2.1 \& } m \text{ is not initial in } C) .
\end{aligned}$$

- Suppose that n' does not occur in the P-view $\ulcorner u \upharpoonright X \urcorner$, then n' lies underneath a PO arc occurring in $\ulcorner u \upharpoonright X \urcorner$. We denote this arc by $\beta\text{-}\alpha$ where β and α denote the arc's nodes. We have:



with $\text{ord}_X \alpha \leq \text{ord}_X m$ (by P-i.j. of $u \upharpoonright X$).

A. Suppose α is an external move then so is β . Indeed, if $X = B, C$ and $\alpha \in \bullet_C$ then α can only point to another move in C and if $X = A, B$ and $\alpha \in \bullet_A$ then since α is an O-move in A, B , it is not initial in A and therefore its justifier must also be in A .

Then instanting Lemma 6.2.7 with $n \leftarrow n'$ gives us $n' \notin \ulcorner u \upharpoonright A, C \urcorner$.

B. Suppose α is a B -move then necessarily so is β . Indeed, if $X = A, B$ then $\alpha \in B$ can only point to a move in B , and if $X = B, C$ then since α is an O-move in the game B, C it is not initial in B and therefore its justifier must also be in B .

Now suppose that $n' \in \ulcorner u \upharpoonright A, C \urcorner$, then by Lemma 6.2.12 (with $\delta, \gamma \leftarrow \alpha, n'$) we have $n' \in \ulcorner u_{\leq \alpha} \upharpoonright A, C \urcorner$. By the 3rd and 4th hypothesis, $u_{\leq \alpha} \upharpoonright X$ is P-i.j. and we can use Lemma 6.2.11 on $u_{\leq \alpha}$:

$$\begin{aligned}
 \text{ord}_{A \multimap C} n' &\leq \text{ord}_Y \alpha && \text{(Lemma 6.2.11 with } u \leftarrow u_{\leq \alpha} \text{)} \\
 &= \text{ord}_X \alpha && \text{(Lemma 6.2.1 \& } \alpha \text{ non initial in } B \text{)} \\
 &\leq \text{ord}_X m && (u \upharpoonright X \text{ is P-i.j.)} \\
 &= \text{ord}_{A \multimap C} m && \text{(Lemma 6.2.1 \& } m \text{ is not initial in } C \text{)} \square
 \end{aligned}$$

Linear composition

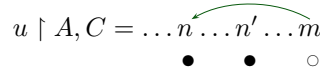
Proposition 6.2.4 (Linear composition). *Let $\sigma : A \multimap B$ and $\mu : B \multimap C$ be two well-bracketed (P-visible) strategies then*

$$(I) \ \sigma \text{ closed P-i.j.} \wedge \mu \text{ P-i.j.} \implies \sigma; \mu \text{ P-i.j.};$$

$$(II) \ \sigma, \mu \text{ closed P-i.j.} \implies \sigma; \mu \text{ closed P-i.j.}$$

Proof. Since well-bracketing is preserved by strategy composition [AMJ94, Proposition 2.5], $\sigma; \mu$ is well-bracketed so we can use the definition of P-i.j. from Proposition 6.2.1.

(I) We prove that $\sigma; \mu$ is P-i.j. Let u be a play of the interaction $\sigma \parallel \mu$ ending with an external P-move m justified by n in $\ulcorner u \upharpoonright A, C \urcorner$. Let n' be an external O-move occurring between n and m :



To show that $u \upharpoonright A, C$ is P-incrementally justified, we just need to prove that either $n' \notin \ulcorner u \upharpoonright A, C \urcorner$ or $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap C} m$. Note that if $n' \in \ulcorner u \upharpoonright A, C \urcorner$ then necessarily n' is not initial in C because n occurs before n' in $\ulcorner u \upharpoonright A, C \urcorner$.

Let E denote one of the two external arenas (A or C), X be the corresponding component (i.e., $X = A, B$ if $E = A$ and $X = B, C$ if $E = C$) and Y denote the other component.

1) Suppose m and n are two external moves in E .

1.a) Suppose $n' \in E$.

This situation is handled by Lemma 6.2.13: we have either $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap C} m$ or $n' \notin \ulcorner u \upharpoonright A, C \urcorner$.

1.b) Suppose $n' \notin E$.

If $n' \in \ulcorner u \upharpoonright A, C \urcorner$, then by Lemma 6.2.11 with $X \leftarrow Y$ we have $\text{ord}_{A \multimap C} n' \leq \text{ord}_X m$ and since m is not initial in C , Lemma 6.2.1 gives $\text{ord}_X m = \text{ord}_{A \multimap C} m$, thus $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap C} m$.

2) Suppose $m \in A$ and $n \in C$.

Then m is an initial move in A pointing to a \bullet -move b_0 initial in B which in turn points to the \bullet_C -move n initial in C .

This situation differs from the previous case because the justifier of m in the game A, C differs from its justifier in A, B (see Sec. 2.3.2.6 for the definition of projection on the overall component A, C), thus it is not guaranteed that m 's justifier in A, C occurs before n' so we cannot use Lemma 6.2.11.

Let's assume that $n' \in \ulcorner u \upharpoonright A, C \urcorner$ and prove that we necessarily have $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap C} m$.

- Suppose n' occurs before b_0 . (Thus we cannot use Lemma 6.2.11). Up to now we have only used the fact that σ and μ are P-i.j. The assumption that σ is *closed* P-i.j. now becomes crucial.

Since $n' \in \ulcorner u \upharpoonright A, C \urcorner$ and $b_0 \in \ulcorner u \upharpoonright B, C \urcorner$, applying Lemma 6.2.12 with $X \leftarrow B, C$ and $\delta, \gamma \leftarrow b_0, n'$ gives $n' \in \ulcorner u_{\leq b_0} \upharpoonright A, C \urcorner$. This allows us to apply Lemma 6.2.11 on $u_{\leq b_0}$:

$$\begin{aligned} \text{ord}_{A \multimap C} m &= \text{ord}_A m \geq \text{ord}_B b_0 && (u \upharpoonright A, B \text{ is closed P-i.j., } m \text{ is initial in } A) \\ &= \text{ord}_{B \multimap C} b_0 \\ &\geq \text{ord}_{A \multimap C} n' && (\text{Lemma 6.2.11 on } u_{\leq b_0} \text{ with } X \leftarrow A, B) . \end{aligned}$$

- Suppose n' occurs after b_0 (and necessarily before m).

a. Suppose $n' \in C$. m 's justifier occurs before n' in u thus by Lemma 6.2.11 we have $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap B} m = \text{ord}_{A \multimap C} m$.

b. Suppose $n' \in A$. Since $n' \in \ulcorner u \upharpoonright A, C \urcorner$, by Lemma 6.2.13 with $X \leftarrow A, B$ and $(n, n', m) \leftarrow (b_0, n', m)$ we have $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap C} m$.

(Note that here we could not use Lemma 6.2.11 on u because m and n' are both played in A . Also, if A has a single initial move then n' is necessarily hereditarily enabled by the initial move m , thus we can immediately conclude that $\text{ord}_{A \multimap C} n' \leq \text{ord}_{A \multimap C} m$, but this argument does not work in the general case.)

(II) We now show that $\sigma; \mu$ is closed P-i.j. provided that both σ and μ are. Take a play $sm \in \sigma; \mu$ such that m is initial in A and let n be the initial move of C justifying m . Let $u \in \sigma \parallel \mu$ be the uncovering of sm ($sm = u \upharpoonright A, C$) and b_0 be the initial B -move justifying m in u . We have:

$$\begin{aligned} \text{ord}_A m &\geq \text{ord}_B b_0 && (u \upharpoonright A, B \in \sigma \text{ is closed P-i.j.}) \\ &\geq \text{ord}_C n && (u_{\leq b_0} \upharpoonright B, C \in \mu \text{ is closed P-i.j.}) \quad \square \end{aligned}$$

Remark: The second part of the proposition only gives a *sufficient* condition for $\sigma; \mu$ to be closed P-i.j. It is possible that $\sigma; \mu$ is closed P-i.j. although μ is not.

Tensor product

Given two strategies $\sigma : A \multimap B$ and $\tau : C \multimap D$, their tensor product is denoted $\sigma \otimes \tau : A \otimes B \multimap C \otimes D$ where $A \otimes B$ denotes the tensor product of the games A and B (See Sec. 2.3.4.1).

Proposition 6.2.5. *If $\sigma : A \multimap B$ and $\tau : C \multimap D$ are P-i.j. (resp closed P-i.j.) then so is $\sigma \otimes \tau$.*

Proof. By establishing the state diagram of the game $A \otimes C \multimap B \otimes D$ one can show easily that only player O can switch between the subgames $A \multimap B$ and $C \multimap D$. Consequently, in the P-view of a play of the game $A \otimes C \multimap B \otimes D$, all the moves are played in the same subgame (i.e., all in $A \multimap B$ or all in $C \multimap D$). Hence if the last move of a play m is played in $A \multimap B$ then $\ulcorner s \upharpoonright A, B \urcorner = \ulcorner s^\top \upharpoonright A, B \urcorner = \ulcorner s^\top \urcorner$ (and conversely if m is played in $C \multimap D$). The result follows immediately. \square

Pairing and projection

Given two strategies $\sigma : C \multimap A$ and $\tau : C \multimap B$, let $\langle \sigma, \tau \rangle : C \multimap A \& B$ denote the pairing strategy as defined in Sec. 2.3.4.3 where $A \& B$ denotes the product of the games A and B .

Proposition 6.2.6 (Pairing).

- (i) If $\sigma : C \multimap A$ and $\tau : C \multimap B$ are *P-i.j.* (resp. *closed P-i.j.*) then so is $\langle \sigma, \tau \rangle$;
- (ii) For any objects A and B , the projections $\pi_1 : A \times B \multimap A$ and $\pi_2 : A \times B \multimap B$ are *closed P-i.j.*

The proof is immediate.

Promotion

Let s be a play. We call **thread** a maximal subsequence of s constituted of moves that are hereditarily justified by the same occurrence of an initial move. For any move m occurring in s , there is only one thread in s containing it, this thread is called the **thread of m** .

Recall that the promotion $\sigma^\dagger : !A \multimap !B$ of a strategy $\sigma : !A \multimap B$, for two well-opened games A and B , is given by:

$$\sigma^\dagger = \{s \in L_{!A \multimap !B} \mid \text{for all initial } m \text{ in } B, s \upharpoonright m \in \sigma\}$$

Since B is well-opened, plays of σ are constituted of a single thread initiated by some initial B -move. Plays of σ^\dagger however, are interleaves of potentially infinitely many single-threaded plays of σ . One can show easily, using the visibility condition, that the thread of a P -move is always the same as the thread of the preceding O -move. Consequently, the P -view of a play is equal to the P -view of the current thread: if the current thread of a play s is opened by an initial move $b \in B$ then $\lceil s \rceil = \lceil s \upharpoonright b \rceil = \lceil s \rceil \upharpoonright b$.

The state of the game is given by an infinite sequence of symbols in $\{O, P\}$, each element of the sequence indicating who is to play in the corresponding thread. The diagram on Fig. 6.2 illustrates how the state changes as a play of σ^\dagger unfolds. The initial state of the game is O^ω —an infinite sequence of O 's—which indicates that O is to play in all the threads. When O plays an initial move in B , it “opens” a new thread so the state of the game becomes $O^k P O^\omega$ where k is the index of the thread being opened. By alternation, P now has to play. His move must be played in a thread already opened by O and in which P is to play; only one thread is in such state: the k th one. Hence after P 's move we are back to state O^ω .

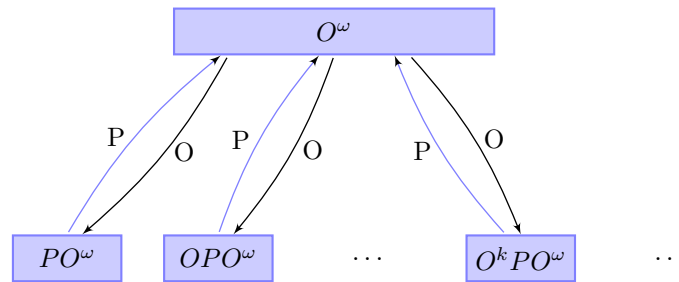


Figure 6.2: State diagram for plays of σ^\dagger .

Proposition 6.2.7 (Promotion). *If A and B are two well-opened games and $\sigma : !A \multimap B$ is a well-bracketed P -i.j. strategy then σ^\dagger is also well-bracketed and P -i.j. Furthermore if σ is closed P -i.j. then so is σ^\dagger .*

Proof. σ^\dagger is well-bracketed [AMJ94, Proposition 2.10.]. For P -incremental justification, the result is a direct consequence of the fact that the P -view of a play in σ^\dagger is equal to the P -view of the current thread. For closed P -incremental justification, the result is immediate. \square

Composition

We recall that the composite of $\sigma : !A \multimap B$, and $\mu : !B \multimap C$ in the co-Kleisli category of games \mathcal{C} , written $\sigma \circledcirc \mu$, is defined as:

$$\sigma \circledcirc \mu = \sigma^\dagger ; \mu .$$

From propositions 6.2.4 and 6.2.7 we obtain:

Proposition 6.2.8. *Let A and B be two well-opened games. Let $\sigma : !A \multimap B$ and $\mu : !B \multimap C$ be two well-bracketed strategies then:*

- (i) *If σ is closed P-i.j. and μ is P-i.j. then $\sigma \circledcirc \mu : !A \multimap C$ is also P-i.j.;*
- (ii) *If σ and μ are closed P-i.j. then so is $\sigma \circledcirc \mu : !A \multimap C$.*

6.2.7 Categories of closed P-i.j. strategies

We define the category of closed P-incrementally justified strategies as follows:

- Objects: games (as defined in Sec. 2.3.2.2),
- Morphisms $\sigma : A \multimap B$: closed P-i.j. strategies for $A \multimap B$,
- Composition: the linear strategy composition.

The results of the previous section show that this is indeed a monoidal category. It is not monoidal closed, however. Indeed, recall that a P-i.j. strategy $\sigma : A \multimap B$ is *closed* P-i.j. if some condition on the initial A -moves occurring in the plays is met. In particular if A has no initial move, σ is necessarily closed P-i.j. Consequently the isomorphic strategy on the game $I \multimap (A \multimap B)$ obtained by *currying* is closed P-i.j. although σ itself is not necessarily closed P-i.j. Take for instance the two simply typed terms $\vdash_{\text{st}} \lambda x^o y^o . y$ and $y : o \vdash_{\text{st}} \lambda x^o . y$. These two terms have isomorphic denotations in \mathcal{G} . But the denotation of the first term is closed P-i.j. while the second is only P-i.j.

Intentional category

We define the *intentional category* \mathcal{I} as the co-Kleisli category of the category defined above.

Proposition 6.2.9. *\mathcal{I} is a sub-ICC of \mathcal{C} .*

Proof. The objects of \mathcal{I} are exactly those of \mathcal{C} . The morphisms of \mathcal{I} are a subclass of morphisms of \mathcal{C} . For every object A , the identity strategy id_A is closed P-i.j. For every pair of morphisms in \mathcal{I} the composite is also in \mathcal{I} by Prop. 6.2.8. Thus \mathcal{I} is a lluf subcategory of \mathcal{C} . By Prop. 6.2.6, projections are closed P-i.j., and closed P-i.j. strategies are closed under pairing. Because of Lemma 6.2.4(i), the incremental evaluation maps are closed P-i.j., and the closed P-i.j. strategies are closed under incremental currying. Hence \mathcal{I} is a sub-ICC of \mathcal{C} . \square

This category will be used to give the intentional game model of safe PCF and safe IA. We write \mathcal{I}_{ib} , \mathcal{I}_b and \mathcal{I}_i to denote its lluf subcategories of innocent, well-bracketed and innocent and well-bracketed strategies respectively.

Extensional category

Let \lesssim denote the usual intrinsic preorder of the category \mathcal{C} (See Sec. 2.3.4.6). The preorder $\lesssim_{\mathcal{I}}$ on morphisms of the category \mathcal{C} is defined similarly to \lesssim except that the test strategy α ranges over the morphisms of the subcategory \mathcal{I} only: for $\sigma, \mu \in \mathcal{C}(I, A)$,

$$\sigma \lesssim_{\mathcal{I}} \tau \iff \forall \alpha \in \mathcal{I}(A, \Sigma). \sigma \circledcirc \tau = \top \implies \tau \circledcirc \alpha = \top .$$

The *intrinsic preorder* in \mathcal{I} , also written $\lesssim_{\mathcal{I}}$, is defined as the restriction of $\lesssim_{\mathcal{I}}$ to the morphisms of the category \mathcal{I} . Abramsky et al. [AMJ94] proved that \lesssim is a CCC precongruence for \mathcal{C} . The same proof shows that $\lesssim_{\mathcal{I}}$ is also a CCC precongruence for \mathcal{C} . Consequently by Lemma 6.1.3, the *extensional category* $\mathcal{I}/\lesssim_{\mathcal{I}}$ is a rational ICC.

Interpretation

By Prop. 6.1.2, we have that the ICCs \mathcal{I} and $\mathcal{I}/\lesssim_{\mathcal{I}}$ both provide a model of the safe lambda calculus, and the rational ICCs \mathcal{I}_{ib} and $\mathcal{I}_{ib}/\lesssim_{\mathcal{I}_{ib}}$ of innocent well-bracketed closed P-i.j. strategies both provide a model of safe PCF.

6.3 Interpretation in the standard game model

In Chapter 5 we have shown using a syntactic argument based on the theory of traversals, that in the standard game model, safe lambda terms are denoted by P-i.j. strategies. We now reprove this result using a semantic argument based on the result of the previous section.

This result is not surprising since we already know by Proposition 6.1.2 that the ICC category \mathcal{I} of closed P-i.j. strategies provides a model of the safe lambda calculus, safe PCF, and safe IA.

6.3.1 Safe lambda calculus with product

Proposition 6.3.1. *In the standard game model of the simply typed lambda calculus with product, safe terms are denoted by closed P-i.j. strategies.*

Proof. We show by induction on the formation rules that (1) almost safe terms are denoted by P-i.j. strategies; (2) safe terms are denoted by *closed* P-i.j. strategies.

- (var) $\llbracket x : A \vdash_s x : A \rrbracket$ is the identity strategy id_A which is closed P-i.j.
- (wk) Take $\Gamma \subset \Delta$ and suppose $\llbracket \Gamma \vdash_s s : A \rrbracket$ is closed P-i.j. Up to a retagging of the moves, the two strategies $\llbracket \Delta \vdash_s s : A \rrbracket$ and $\llbracket \Gamma \vdash_s s : A \rrbracket$ are isomorphic. Hence $\llbracket \Delta \vdash_s s : A \rrbracket$ is P-i.j. It is also closed P-i.j. since none of the new initial moves introduced by Δ occurs in any play of the strategy.
- (\times), (π_1) and (π_2): The result follows from the I.H. and Proposition 6.2.6.
- (δ): It follows from the I.H.
- (app_{as}) Suppose that $\Gamma \vdash_{app} t_0 t_1 \dots t_n : B$ with $\Gamma \vdash_s t_0 : (A_1, \dots, A_n, B)$ and $\Gamma \vdash_s t_i : A_i$ for $i \in \{1..n\}$. By the I.H., for $i \in \{0..n\}$ the strategy $\llbracket t_i \rrbracket$ is closed P-i.j. We then have $\llbracket t_0 t_1 \dots t_n \rrbracket = \langle \llbracket t_0 \rrbracket, \llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket \rangle ; ev^n$ where ev^n is the n -parameter evaluation strategy. By Proposition 6.2.6 the strategy $\langle \llbracket t_0 \rrbracket, \llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket \rangle$ is closed P-i.j. Since the evaluation map ev^n is P-i.j. (but not necessarily closed P-i.j.), by Proposition 6.2.4.I. $\llbracket \Gamma \vdash_s t_0 t_1 \dots t_n : B \rrbracket$ is P-i.j.
- (app) Terms formed with this rule can also be formed with the rule (app_{as}), therefore by the previous case the denotation of the term formed is P-i.j. By the side-condition of the rule, all the prime sub-types of Γ have order greater than the order of the term, therefore by Lemma 6.2.4(ii), $\llbracket \Gamma \vdash_s t_0 t_1 \dots t_n : B \rrbracket$ is *closed* P-i.j.
- (abs): By the I.H., the premise of the rule has a P-i.j. denotation. The denotation of the term in the conclusion of the rule is isomorphic, up to currying, to the denotation of the premise. Therefore it is also P-i.j. And by the side-condition and Lemma 6.2.4(ii) this implies that it is *closed* P-i.j. \square

6.3.2 Safe PCF

Proposition 6.3.2. *In the standard game model of PCF, safe terms are denoted by closed P-incrementally justified strategies.*

Proof. We first prove the result for PCF_1 —the fragment of PCF containing terms of the form $\Omega_A = Y(\lambda x^A. x)$ but where no other use of Y is allowed [AM98b]. The proof is by structural induction over the structure of the term:

- The strategy $\llbracket \Omega_A \rrbracket = \perp$ is clearly closed P-i.j.;
- The functional rules are treated the same way as in the corresponding proof for the safe lambda calculus;

• For the arithmetic rules, we observe that the strategies *succ*, *pred* and *cond* are all closed P-i.j. The fact that pairing and strategy composition preserve closed P-incremental justification permits us to conclude.

We now lift the result to full PCF using the technique of *syntactic approximant* [AM98b]. We have [AM98b, lemma 16]:

$$\llbracket M \rrbracket = \bigcup_{n \in \omega} \llbracket M_n \rrbracket$$

where M_n is the PCF_1 term obtained from M by replacing each subterm of the form YN with $Y^n N_n$, and $Y^n F$ denotes the n th approximant of YF . Since the M_n s are PCF_1 terms, by the previous result each $\llbracket M_n \rrbracket$ is closed P-i.j. and since closed P-incremental justification is clearly a continuous property, $\llbracket M \rrbracket$ is also closed P-i.j. \square

6.3.3 Safe Idealized Algol

We now extend the game-semantic interpretation to safe IA. The constants of IA are all denoted by closed P-incrementally justified strategies:

Lemma 6.3.1.

- (i) The strategy denotations of the IA constants **skip**, **assign**, **deref**, **mkvar**, **seq_{exp}**, and **seq_{com}** are all closed P-i.j.;
- (ii) The memory-cell strategy $cell : I \multimap !\mathbf{var}$ is closed P-i.j.

Proof. (i) A quick inspection at the view function of these denotations (as defined in Sec. 2.3.6) reveals that they are indeed all closed P-i.j.

(ii) Since the game **var** does not contain any P-question, any strategy on the game $I \multimap !\mathbf{var}$ is P-i.j. (and therefore also closed P-i.j.). \square

Our game-semantic analysis of safe PCF immediately extends to strongly safe IA:

Proposition 6.3.3. *Strongly safe IA terms are denoted by closed P-i.j. strategies.*

Proof. The proof is an adaptation of the proof for Safe PCF. We first show that the result holds for the fragment of *strongly safe IA* in which the only allowed uses of Y are in terms of the form Ω . This is done by induction over the structure of the term: For the functional and arithmetic rules, the proof is the same as for Safe PCF. For the imperative rules, the result follows from the fact that IA constants are denoted by closed P-i.j. strategies (Lemma 6.3.1(i)) and because tensor product and composition both preserve closed P-incremental justification. For the block-allocation construct, the result follows from the fact that *cell* is closed P-i.j. (Lemma 6.3.1(ii)) and that pairing and strategy composition both preserve closed P-incremental justification.

The result is then lifted to the whole of strongly safe IA using the technique of syntactic approximants as in the PCF case. \square

We now want to extend this result to safe IA. This turns out to be slightly more difficult than for the strongly-safe fragment. Indeed, in safe IA the safety restriction only constrains variables from the Γ -context (*i.e.*, those that are bound by a λ -abstraction). The fact that Ξ -variables are not constrained is reflected in the semantics. For instance the denotation of the safe *split*-term $\emptyset | x : \mathbf{var} \vdash_s \lambda f^{\mathbf{exp} \rightarrow \mathbf{exp}}. \mathbf{deref} \ x$ is not closed P-i.j.

We show, however, that safe *split*-terms are denoted by strategies in which all the plays are closed P-i.j. except those containing moves from the Ξ -context. Consequently, by “abstracting” Ξ -variables using the constructs **mkvar** or the block-declaration **new**, we eliminate the plays that are not closed P-i.j. Hence since safe IA terms are the *semi-closed* *split*-terms (*i.e.*, with an empty Ξ -component), this implies that their denotation is closed P-i.j.

Definition 6.3.1 (P-i.j. modulo \mathfrak{M}). Let σ be a strategy on some game A and \mathfrak{M} be a set of moves. We say that σ is *P-incrementally justified modulo \mathfrak{M}* iff every even-length play in σ ending with a question that is not in \mathfrak{M} is P-i.j. Similarly we say that σ is *closed P-i.j. modulo \mathfrak{M}* iff all such plays are *closed P-i.j.*

Hence a strategy is P-i.j. if and only if it is P-i.j. modulo \emptyset .

The common operations on strategies preserve the property of being P-incremental justification modulo a set of moves:

Lemma 6.3.2 (Composition). *Let $\sigma : A \rightarrow B$ and $\mu : B \rightarrow C$. Let \mathfrak{M} be any set of moves initial in A . If σ is closed P-i.j. modulo \mathfrak{M} and μ is P-i.j. (resp. closed P-i.j.) then $\sigma \circ \mu$ is P-i.j. (resp. closed P-i.j.) modulo \mathfrak{M} .*

Proof. We observe that in the proof of compositionality for closed P-i.j. strategies, to show that a play $u \upharpoonright A, C$ of $\sigma; \mu$ is P-i.j. we did not use the fact that *every* play of σ is P-i.j., but only that $u \upharpoonright A, B$ (resp. $u \upharpoonright B, C$) is P-i.j. and all the prefixes of $u \upharpoonright A, B$ and $u \upharpoonright B, C$ ending with a non-initial B -move are P-i.j. Thus the same proof can be used to show that a play $u \upharpoonright A, C$ ending with a move not in \mathfrak{M} is P-i.j. \square

Lemma 6.3.3 (Tensor product). *Let $\sigma : A \multimap B$ and $\tau : C \multimap D$. Let \mathfrak{M}_A and \mathfrak{M}_C be two sets of moves initial in A and C respectively.*

1. *If σ and τ are P-i.j. modulo \mathfrak{M}_A and modulo \mathfrak{M}_C respectively then $\sigma \otimes \tau$ is P-i.j. modulo $\mathfrak{M}_A \cup \mathfrak{M}_C$;*
2. *If σ and τ are closed P-i.j. modulo \mathfrak{M}_A and modulo \mathfrak{M}_C respectively then $\sigma \otimes \tau$ is closed P-i.j. modulo $\mathfrak{M}_A \cup \mathfrak{M}_C$.*

Lemma 6.3.4 (Pairing). *Let $\sigma : C \multimap A$, $\tau : C \multimap B$, and \mathfrak{M}_C be a sets of moves initial in C .*

- (i) *If σ and τ are P-i.j. modulo \mathfrak{M}_C then so is $\langle \sigma, \tau \rangle$;*
- (ii) *If σ and τ are closed P-i.j. modulo \mathfrak{M}_C then so is $\langle \sigma, \tau \rangle$.*

The proof of the two previous lemmas is an easy adaptation of the proofs of their counterpart for P-i.j. strategies.

Lemma 6.3.5. *Let $\tau : I \rightarrow C_2$, $\sigma : C_1 \otimes C_2 \rightarrow B$ and \mathfrak{M} be any set of moves initial in $C_1 \otimes C_2$. If τ is P-i.j. and σ is P-i.j. (resp. closed P-i.j.) modulo \mathfrak{M} then $(id_{C_1} \otimes \tau) \circ \sigma$ is P-i.j. (resp. closed P-i.j.) modulo $\mathfrak{M} \cap C_1$.*

Proof. Let $D = C_1 \otimes C_2$. Let $u \in \text{Int}(C_1, D, B)$ be a non-empty interaction play of $\mu = (id_{C_1} \otimes \tau)^\dagger \parallel \sigma$, and m denote the last play of u . We need to show that if m does not belong to \mathfrak{M} then $u \upharpoonright C_1, B$ is P-incrementally justified.

Suppose $m \in C_1 \setminus \mathfrak{M}$. Let d be the initial D -move hereditarily justifying m , then by definition of μ we have $u \upharpoonright C_1, D, d \in id_{C_1}$ which implies that $u \upharpoonright C_1, B = u \upharpoonright D, B$. But u is an interaction sequence therefore $u \upharpoonright D, B \in \sigma$, and since σ is P-i.j. modulo \mathfrak{M} this implies that $u \upharpoonright C_1, B$ is P-incrementally justified.

Suppose $m \in B$ then necessarily its justifier also occurs in B . By definition of u , the play $u \upharpoonright D, B$ belongs to σ which is P-i.j. modulo \mathfrak{M} . Since m belongs to B it cannot be in \mathfrak{M} therefore u is P-i.j. Furthermore, since τ is P-i.j., so is $(id_{C_1} \otimes \tau)^\dagger$ therefore the play $u \upharpoonright C_1, D$ and all its prefixes are P-i.j. Hence we can apply Lemma 6.2.13 with $X \leftarrow D, B$ and $Y \leftarrow C_1, D$ which shows that $u \upharpoonright C_1, B$ is P-i.j. \square

Lemma 6.3.6. *Let $\text{mkvar} : B \rightarrow C$ be the denotation of the `mkvar` construct where $B = (\text{exp}^1 \rightarrow \text{com}) \times \text{exp}$ and $C = \text{var}$. If $\sigma : A \rightarrow B$ is a closed P-i.j. strategy modulo $\mathcal{M}_A \cup \llbracket \text{exp}^1 \rrbracket$ for some set \mathfrak{M}_A of initial A -moves then $\sigma; \text{mkvar}$ is closed P-i.j. modulo \mathfrak{M}_A .*

Proof. Let u be an interaction sequence such that $u \upharpoonright A, C$ ends with a P-question that is not in \mathfrak{M}_A . Then $u \upharpoonright A, B$ and $u \upharpoonright B, C$ are both P-i.j. Let m denote the last move in u and n be its justifier in $u \upharpoonright A, C$. Suppose that an O-move n' occurs in the P-view between n and m . We show that its order is necessarily smaller than that of m . We necessarily have $m \in \circ_A$ because there is no P-question in C .

(a) Suppose that $m \in \circ_A$, $n \in \bullet_A$ and $n' \in \bullet_A$. In general, n' does not necessarily appear in the P-view $\ulcorner u \upharpoonright A, B \urcorner$ (See proof of compositionality.) In the present case, however, this case never happens. Indeed, as noted in the proof of Lemma 6.2.13, this would imply that n' lies underneath a \bullet - \bullet -arc. But this is not possible since the only \bullet -move in B is an initial move. Thus n' occurs in $\ulcorner u \upharpoonright A, B \urcorner$ and since $u \upharpoonright A, B$ is P-i.j. this imply that n' has order smaller than m .

(b) Suppose that $m \in \circ_A$, $n \in \bullet_A$ and $n' \in \bullet_C$. Take $Y = A, B$ and $X = B, C$. We have that $u \upharpoonright Y$ is P-i.j. and since $mkvar$ is a P-i.j. strategy, for all B -move b occurring in u , $u_{\leq b} \upharpoonright X$ is P-i.j. Thus we can apply Lemma 6.2.11 which shows that $\text{ord}_{A \rightarrow C} n' \leq \text{ord}_{A \rightarrow C} m$.

(c) Suppose $m \in \circ_A$, $n \in \bullet_C$. Then in A, B , the move m is justified by a \bullet -move b_0 itself justified by n in B, C . By definition of the strategy $mkvar$, n and b_0 are in fact consecutive moves in u , thus n' necessarily occurs after b_0 . If $n' \in \bullet_C$ then we conclude with Lemma 6.2.11 as in (b) that $\text{ord}_{A \rightarrow C} n' \leq \text{ord}_{A \rightarrow C} m$. Otherwise $n' \in \bullet_A$, and we conclude as in (a).

Hence $u \upharpoonright A, C$ is P-i.j. It is *closed* P-i.j. because both $u \upharpoonright A, B$ and $u \upharpoonright B, C$ are. \square

Example 6.3.1. The unsafe term

$$f : (\text{exp} \rightarrow \text{exp}) \rightarrow \text{com} \vdash \lambda x. f(\lambda y. \underline{x}) \equiv M : \text{exp}^1 \rightarrow \text{com}$$

is denoted by a strategy $\llbracket M \rrbracket$ that is closed P-i.j. modulo $\llbracket \text{exp}^1 \rrbracket$. But the term $\text{mkvar } M 0 : \text{var}$ is denoted by the strategy $\langle \llbracket M \rrbracket, 0 \rangle; mkvar$ which is closed P-i.j.

Given a safe split-term $\Gamma | \Xi \vdash_s M : A$, we write $\llbracket \Gamma | \Xi \vdash_s M : A \rrbracket$ to refer to $\llbracket \Gamma, \Xi \vdash M : A \rrbracket$, the game denotation of the corresponding IA split-term. For any game A we write $\text{In}(A)$ for the set of initial moves in A .

Proposition 6.3.4. *Let $\Gamma | \Xi \vdash_s M : A$ be a safe IA split-terms. Its denotation $\llbracket \Gamma | \Xi \vdash_s M : A \rrbracket$ is closed P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket)$.*

REMARK 6.3.1 $\text{In}(\llbracket \Xi \rrbracket)$ contains only order-0 questions because the context Ξ contains variables of type **var** and **exp** only.

Proof. We only need to prove the result for terms where the only allowed uses of the Y combinator is in subterms of the form Ω , the result then follows immediately using the syntactic approximants technique and continuity of the “closed P-i.j.” property.

We proceed by induction on the safe IA term. The cases (**var**), (**wk**), (**const**), (**succ**), (**pred**), (**cond**) are the same as for safe PCF.

- (**var^{new}**), (**wk^{new}**) are similar to (**var**) and (**wk**).

- (**seq**), (**assign**), (**deref**) These constants all have closed P-i.j. denotations so the result follows from the I.H., Lemma 6.3.2, Proposition 6.3.4 and 6.3.3.

- (**app**) The premise of the rule is an almost safe split-term (i.e., a consecutive applications of safe terms). By the I.H. all these terms have a denotation that is closed P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket)$. Since the evaluation strategy ev is P-i.j., by Lemma 6.3.2 this implies that the denotation of the split-term being formed is P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket)$. Finally, the side-condition of the rule ensures that it is *closed* P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket)$.

- (**abs**) It follows from the I.H. and because the side condition of the abstraction rules constrains only free variables from the Γ -context.

- (**new**) Let $\sigma = \llbracket \Gamma | \Xi, x : \text{var} \vdash_s M : B \rrbracket$. We have $\llbracket \Gamma | \Xi \vdash_s \text{new } x \text{ in } M : B \rrbracket = (id_{\Gamma, \Xi} \otimes cell) \circ \sigma$ where $cell$ denotes the memory cell strategy on the game $I \rightarrow !\text{var}$. By the I.H. σ is closed P-i.j. modulo $\text{In}(\llbracket \Xi \otimes !\text{var} \rrbracket)$. Instanting Lemma 6.3.5 with $\tau \leftarrow cell$, $C_1 \leftarrow \Gamma \otimes \Xi$ and $C_2 \leftarrow !\text{var}$ gives us the desired result.

- (mkvar) Let $\sigma = \llbracket \Gamma | \Xi \vdash_s \text{mkvar} (\lambda x. M_1) M_2 \rrbracket$. We have $\sigma = \langle \Delta(\sigma_1), \sigma_2 \rangle; \text{mkvar}$ where $\sigma_1 = \llbracket \Gamma | \Xi, x : \text{exp} \vdash_s M_1 : \text{com} \rrbracket$ and $\sigma_2 = \llbracket \Gamma | \Xi \vdash_s M_2 : \text{exp} \rrbracket$. By the I.H. σ_1 is closed P-i.j. modulo $\text{In}(\llbracket \Xi, x : \text{exp} \rrbracket)$ and σ_2 is closed P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket)$ therefore the strategy $\langle \Delta(\sigma_1), \sigma_2 \rangle : \llbracket \Gamma \times \Xi \rightarrow (\text{exp}^1 \rightarrow \text{com}) \times \text{exp} \rrbracket$ is closed P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket \cup \llbracket \text{exp}^1 \rrbracket)$. Hence by Lemma 6.3.6, σ is closed P-i.j. modulo $\text{In}(\llbracket \Xi \rrbracket)$. \square

By definition, safe IA terms are the semi-closed safe split-terms, hence:

Proposition 6.3.5. *In the standard game model of IA, safe terms are denoted by closed P-i.j. strategies.*

6.4 O-incremental justification

We define *O-incremental justification* as the dual of P-incremental justification:

Definition 6.4.1.

- (i) A play sm of *odd* length is said to be **O-incrementally justified**, or *O-i.j.* for short, if m points to the last unanswered P-question in $\lceil s \rceil$ with order strictly greater than $\text{ord } m$.
- (ii) A strategy σ is said to be **O-incrementally justified**, if all plays in σ ending with an O-question are O-incrementally justified.

This corresponds to the constraint that has to be imposed on the Opponent to reflect the fact that the environment is itself incarnated by a safe term. This duality is similar to the one between O-visibility and P-visibility [Har05, Sec. 3.6].

For any strategy σ , we write $\mathcal{O}(\sigma)$ to denote the largest subset of plays of σ whose odd-length prefixes are all O-i.j. The set $\mathcal{O}(\sigma)$ is obtained by removing all the plays containing O-moves that are not incrementally justified. It defines a strategy that mimics the strategy σ as long as the Opponent plays incrementally and does not answer otherwise.

Lemma 6.4.1. *Let $\sigma : A$ and $\alpha : A \rightarrow o$ be two strategies.*

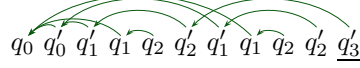
In the composition $\sigma; \alpha$, the P-i.j. plays of σ interact only with O-i.j. plays of α , and the O-i.j. plays of σ interact only with P-i.j. plays of α .

Proof. Let $\sigma : A$ and $\alpha : A \rightarrow o$ be two strategies, and q be the initial move of the game $\llbracket A \rightarrow o \rrbracket$. For every $s \in L_A$ we have $qs \in L_{A \rightarrow o}$. P-moves and O-moves in $\llbracket A \rrbracket$ become O-moves and P-moves in $\llbracket A \rightarrow o \rrbracket$ respectively. Hence P-views of plays in A correspond to O-views in $A \rightarrow o$: $q^\top s^\top A = \perp qs \perp_{A \rightarrow o}$. Now take an interaction sequence $u = qv \in \sigma \parallel \alpha$. We have $u \upharpoonright (A \rightarrow o) = (qv) \upharpoonright (A \rightarrow o) = q(v \upharpoonright A)$. Hence if $u \upharpoonright A = v \upharpoonright A$ is P-i.j. then by the previous remark, $u \upharpoonright (A \rightarrow o)$ is O-i.j. The proof of the second part is symmetrical. \square

Lemma 6.4.2. *In an order-3 well-opened game all the legal positions are O-i.j.*

Proof. Let A be an order-3 well-opened game. Take a play s in σ ending with a question move q . We prove by induction on s that if q is a non-initial O-move then there is a single P-move in $\perp s \perp$ with order $> \text{ord } q$. We do a case analysis on the level of q . We recall that $\text{ord } q + \text{level } q \leq \text{ord } A$. Since q is a non-initial O-move, we necessarily have $\text{level } q = 2$. Let q' denote the P-move preceding q in s . Suppose that $\text{level } q' = 1$ then q' is justified by an occurrence of the initial A-move q_0 . Since A is well-opened, s contains only one occurrence of q_0 and therefore we have $\perp s \perp = q_0 \cdot q' \cdot q$. Thus by O-visibility, q necessarily points to q' therefore $\text{ord } q' > \text{ord } q$, thus since q' is the only P-move occurring in the O-view, it is also the only P-move with order greater than $\text{ord } q$. Otherwise we have $\text{level } q' = 3$. Thus $\text{ord } q' \leq \text{ord } A - \text{level } q' = 0$ and q' is justified by some O-move q'' of level 2. We have $\perp s \perp = \perp s \leq q'' \perp \cdot q' \cdot q$. Thus we can conclude using the I.H. on $s \leq q''$ on the fact that $\text{ord } q' = 0 < \text{ord } q$. Hence s is necessarily O-i.j. \square

This lemma does not hold anymore at order 4. For instance the identity strategy $id_A : A \rightarrow A$ on the order-3 game $A = \llbracket ((o^3 \rightarrow o^2) \rightarrow o^1) \rightarrow o^0 \rrbracket$ contains the following play which is not O-i.j.:



where we write m' to denote moves from the left copy of A .

Corollary 6.4.1. *Let σ, μ be two strategies from $\mathcal{C}(I, A)$ where A is an order-3 game. Then*

$$\sigma \lesssim \mu \iff \sigma \lesssim_{\mathcal{I}} \mu .$$

Proof. Let $\alpha : A \rightarrow o$ be a test strategy. By Lemma 6.4.2, σ and μ are necessarily O-i.j. Thus by Lemma 6.4.1, the plays of σ, μ can only interact with P-i.j. plays from α . Hence $\sigma; \alpha = \sigma; \mathcal{P}(\alpha)$ and $\mu; \alpha = \mu; \mathcal{P}(\alpha)$. Therefore by definition of the intrinsic preorders we have $\sigma \lesssim \mu$ iff $\sigma \lesssim_{\mathcal{I}} \mu$. \square

6.5 Full abstraction

Question: What is a fully abstract model of safe PCF and safe IA?

We already know from the fully-abstract game model of PCF that when the observational preorder is defined with respect to unrestricted (*i.e.*, possibly unsafe) PCF contexts, observational equivalence is captured by equality of the quotiented game denotations. We show here that a similar correspondence holds when observational equivalence is defined with respect to safe contexts only. This further implies a full abstraction result for the fragments of PCF and IA consisting of safe closed terms.

Observational equivalence with respect to safe contexts

We first recall some basic definitions. A *context* is a PCF term containing exactly one free occurrence of a distinguished variable ‘ $-$ ’ called the “hole”. A context is usually denoted by $C[-]$ so that

$$- : A \vdash C[-] : B$$

is a valid PCF term-in-context for some type A and B . For any term M of type A we write $C[M]$ to denote the term obtained by substituting M for the hole using capture-permitting substitution. Due to the possibility of variable capture, this term is not necessarily a valid PCF term. Also it is possible to have $C_1[-] =_{\beta} C_2[-]$ and $C_1[M] \neq_{\beta} C_2[M]$. (For instance take $C_1[-] \equiv (\lambda x^{\text{exp}}. -)0$ and $C_2[-] \equiv (\lambda x^{\text{exp}}. -)\Omega$. Then $C_1[-] =_{\beta} - =_{\beta} C_2[-]$. But $C_1[x] =_{\beta} 0$ and $C_2[x] =_{\beta} \Omega$.)

We write $\text{Trm}(\Gamma, A)$ for the set of terms M such that $\Gamma \vdash M : A$ is derivable in PCF. Terms in $\text{Trm}(\emptyset, \text{exp})$ (*i.e.*, closed PCF terms of base type) are called **PCF program**. For any typing-context Γ and type $A \in \mathbb{T}$ the **program contexts** $\text{Ctx}(\Gamma, A)$ are the PCF contexts $C[-]$ such that for all $M \in \text{Trm}(\Gamma, A)$, the term $C[M]$ is a PCF program.

We write $\text{Trm}_s(\Gamma, A)$ for the set of terms M such that $\Gamma \vdash M : A$ is derivable in safe PCF. We say that a PCF context $C[-]$ is a **safe context** if the judgment

$$- : A \vdash_s C[-] : B,$$

is a valid safe PCF term-in-context. The **safe program contexts** $\text{Ctx}_s(\Gamma, A)$ are the program contexts from $\text{Ctx}(\Gamma, A)$ that are safe contexts.

We now define two notions of observational preorder for PCF:

Definition 6.5.1 (Observational preorders). Let Γ be a typing-context and T be a simple type. Let M and N range over $\text{Trm}(\Gamma, T)$. We write \sqsubseteq to denote the standard observational preorder for PCF terms. This relation on $\text{Trm}(\Gamma, T)$ is defined as:

$$M \sqsubseteq N \stackrel{\text{def}}{=} \forall C[-] \in \text{Ctx}(\Gamma, A). C[M] \Downarrow \implies C[N] \Downarrow .$$

The relation \sqsubseteq_s on $\text{Trm}(\Gamma, T)$ is defined similarly to \sqsubseteq except that contexts range over safe terms only:

$$M \sqsubseteq_s N \stackrel{\text{def}}{=} \forall C[-] \in \text{Ctx}_s(\Gamma, A). C[M] \Downarrow \implies C[N] \Downarrow .$$

We write \cong and \cong_s to denote the reflexive closures of \sqsubseteq and \sqsubseteq_s .

Lemma 6.5.1.

- (i) The relations \sqsubseteq and \sqsubseteq_s are preorders (reflexive and transitive);
- (ii) Consequently \cong and \cong_s are equivalence relations;
- (iii) $\sqsubseteq \subseteq \sqsubseteq_s$.

Proof. Trivial. □

Note that in the definition of \sqsubseteq_s , the program context $C[-]$ ranges in $\text{Ctx}_s(\Gamma, A)$ but it is not required that $C[M]$ and $C[N]$ are themselves safe. When restricted to safe terms, however, $C[M]$ and $C[N]$ are necessarily safe:

Lemma 6.5.2. $M \in \text{Trm}_s(\Gamma, T) \wedge C[-] \in \text{Ctx}_s(\Gamma, T) \implies C[M] \in \text{Trm}_s(\emptyset, \text{exp})$.

Proof. Suppose that $M \in \text{Trm}_s(\Gamma, T)$ and $C[-] \in \text{Ctx}_s(\Gamma, T)$ then in particular, $M \in \text{Trm}(\Gamma, T)$ and $C[-] \in \text{Ctx}(\Gamma, T)$, therefore by definition of a program context we have $C[M] \in \text{Trm}(\emptyset, \text{exp})$.

Plugging a term in the context is done via capture-permitting substitution: $C[M]$ is given by $(C[-])\{M/-\}$. Since both $C[-]$ and M are safe and $C[M]$ is a valid term, by the No-variable-capture lemma (Corollary 3.5.2(ii)) it is syntactically equivalent to perform the standard substitution: $C[M] \equiv (C[-])[M/-]$. Hence by the Substitution Lemma 3.1.6, $C[M]$ is safe. □

Lemma 6.5.3. $M \in \text{Trm}_s(\Gamma, T) \wedge C[-] \in \text{Ctx}_s(\Gamma, T) \implies \llbracket C[M] \rrbracket = \llbracket C[-] \rrbracket; \llbracket M \rrbracket$.

Proof. By the previous lemma, plugging M in $C[-]$ can be done using the capture-permitting substitution therefore $\llbracket C[M] \rrbracket = \llbracket C[-] \rrbracket; \llbracket M \rrbracket$. □

Note that this lemma does not hold for unsafe context. For instance with $C[-] \equiv (\lambda x^{\text{exp}}. -)\Omega$ we have $\llbracket C[-] \rrbracket; \llbracket M \rrbracket = id_A; \llbracket M \rrbracket = \llbracket M \rrbracket$ but $\llbracket C[x] \rrbracket = \perp$.

REMARK 6.5.1 It is possible to define a third notion of observational preorder where the contexts are unrestricted but where we require instead that $C[M]$ and $C[N]$ are safe. This notion of observational preorder differs from \sqsubseteq_s because the safety of $C[M]$ does not necessarily implies that of $C[-]$ (e.g., the context $- : A \vdash \lambda x^A. - : B$ is unsafe although $C[x]$ is safe).

REMARK 6.5.2 Compared to \sqsubseteq , the observational preorder \sqsubseteq_s is a relatively coarse approximation relation for open terms. If we fix a type T then all the open terms of type T containing variables of order at least T will be equated by \sqsubseteq_s . The is because for any such term M , there is no safe context $C[-]$ such that $C[M]$ is closed. Indeed, if $C[M]$ is closed then all the free variables in M must be abstracted in $C[M]$. Take a variable $z \in FV(M)$ verifying $\text{ord } z \geq \text{ord } T$, then the hole in $C[-]$ must appear in a subterm of the form $\lambda z. \dots - \dots$ containing the hole ‘-’. But then this implies that the context is unsafe because the hole, which has order smaller than z , is not abstracted with z . For example, the terms $x : \text{exp} \vdash \text{cond } 0 \ x \ i \equiv M_i : \text{exp}$ for $i \in \mathbb{N}$ are all \cong_s -equivalent, but their closures $N_i \equiv \lambda x^{\text{exp}}. M_i$ are not: $N_i \not\sqsubseteq_s N_j$ for every $i \neq j$.

Proposition 6.5.1 (Computational Adequacy). *Let P be a PCF program. Then*

$$P \Downarrow \iff \llbracket P \rrbracket_C \neq \perp \iff \llbracket P \rrbracket_C \not\approx_I \perp .$$

Proof. The first equivalence is the Computational Adequacy result for PCF [AM97]. Second equivalence: The $\lesssim_{\mathcal{I}_{ib}}$ -equivalence class of \perp contains only the strategy \perp itself. Indeed, suppose that $\sigma \lesssim_{\mathcal{I}_{ib}} \perp$ then for all P-i.j. strategy $\alpha : A \rightarrow \Sigma$ we have $\sigma \circ \alpha = \top \implies \perp \circ \alpha = \top$. But the condition $\perp \circ \alpha = \top$ never holds therefore we necessarily have $\sigma \circ \alpha = \perp$ for all P-i.j. strategy α . In particular, since the identity strategy id_A is P-i.j. we can take $\alpha = id_A$ giving us $\sigma = \sigma \circ id_A = \perp$.

Hence we have $\llbracket P \rrbracket_{\mathcal{C}} \neq \perp$ iff $\llbracket P \rrbracket_{\mathcal{C}} \not\lesssim_{\mathcal{I}_{ib}} \perp$. \square

Proposition 6.5.2 (Inequational soundness). *Let $M, N \in \text{Trm}(\Gamma, T)$. Then:*

$$\llbracket M \rrbracket_{\mathcal{C}} \subseteq \llbracket N \rrbracket_{\mathcal{C}} \implies M \sqsubseteq_s N .$$

Proof. It follows from Inequational soundness in \mathcal{C} [AM97] since \sqsubseteq is a subset of \sqsubseteq_s . \square

Theorem 6.5.1 (Inequational soundness in $\mathcal{C}_{ib}/\lesssim_{\mathcal{I}_{ib}}$). *Let $M, N \in \text{Trm}(\Gamma, T)$. Then:*

$$\llbracket M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}_{ib}} \llbracket N \rrbracket_{\mathcal{C}} \implies M \sqsubseteq_s N .$$

Proof. We first show the result for closed terms: We follow the same argument as the proof of Inequational soundness for PCF [AM97]. Let $M, N \in \text{Trm}(\emptyset, T)$ and suppose that $\llbracket M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}_{ib}} \llbracket N \rrbracket_{\mathcal{C}}$ and that $C[M] \Downarrow$ for some safe context $C[-]$. Then the denotation of $C[-]$ is a P-i.j. strategy $\alpha \in \mathcal{I}(T, \Sigma)$. For any closed term P , the context-substitution $C[P]$ causes no variable capture therefore we have $\llbracket C[P] \rrbracket = \llbracket P \rrbracket \circ \alpha$. Thus by Computational Adequacy we have $\llbracket M \rrbracket \circ \alpha \neq \perp$. But since $\llbracket M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}} \llbracket N \rrbracket_{\mathcal{C}}$ this implies that $\llbracket N \rrbracket \circ \alpha \neq \perp$ which by Computational Adequacy implies $C[N] \Downarrow$ as required.

We now generalize the result to open terms. We first make an observation: For all $C[-] \in \text{Ctx}_s(\Gamma, T)$ and $M \in \text{Trm}(\Gamma, T)$ where $\Gamma = \bar{x} : \bar{A}$ we have:

$$C[M] \Downarrow \iff C[\lambda \bar{x} \bar{A}. M \bar{x}] \Downarrow \iff C'[\lambda \bar{x} \bar{A}. M] \Downarrow$$

where $C'[-]$ is the program context defined as $C'[-] \equiv C[-\bar{x}]$. It is easy to see that this context is necessarily safe: $C'[-] \in \text{Ctx}_s(\Gamma, (\bar{A}, T))$.

Now consider two open terms $M, N \in \text{Trm}(\Gamma, T)$. W.l.o.g. we can assume that $\Gamma = \bar{x} : \bar{A}$ where $\bar{x} = FV(M) \cup FV(N)$. We then have

$$\begin{aligned} \llbracket M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}_{ib}} \llbracket N \rrbracket_{\mathcal{C}} &\iff \Lambda^{|\bar{x}|}(\llbracket \lambda \bar{x} \bar{A}. M \rrbracket_{\mathcal{C}}) \lesssim_{\mathcal{I}} \Lambda^{|\bar{x}|}(\llbracket \lambda \bar{x} \bar{A}. N \rrbracket_{\mathcal{C}}) \\ &\iff \llbracket \lambda \bar{x} \bar{A}. M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}} \llbracket \lambda \bar{x} \bar{A}. N \rrbracket_{\mathcal{C}} \\ &\iff \llbracket \lambda \bar{x} \bar{A}. M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}} \llbracket \lambda \bar{x} \bar{A}. N \rrbracket_{\mathcal{C}} \\ &\iff \forall C'[-] \in \text{Ctx}_s(\Gamma, (\bar{A}, T)). C'[\lambda \bar{x} \bar{A}. M] \Downarrow \implies C'[\lambda \bar{x} \bar{A}. N] \Downarrow \\ &\iff \forall C[-] \in \text{Ctx}_s(\Gamma, T). C[\lambda \bar{x} \bar{A}. M] \Downarrow \implies C[\lambda \bar{x} \bar{A}. N] \Downarrow \quad \text{By Eq. 6.5} \\ &\implies M \sqsubseteq_s N . \quad \square \end{aligned}$$

The **star fragment** of PCF written PCF^* , consists of all the judgements $\Gamma \vdash M : T$ verifying the condition:

$$\forall z : A \in \Gamma. \text{ord } A < \text{ord } T \quad (6.5)$$

abbreviated as “ $\text{ord } \Gamma < \text{ord } T$ ”.

Theorem 6.5.2 (Full abstraction of PCF^* with respect to safe context). *Let $M, N \in \text{Trm}(\Gamma, T)$ be two PCF terms with $\text{ord } \Gamma < \text{ord } T$. Then*

$$\begin{aligned} M \sqsubseteq_s N &\iff \llbracket M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}_{ib}} \llbracket N \rrbracket_{\mathcal{C}} & (i) \\ &\iff \mathcal{O}(\llbracket M \rrbracket_{\mathcal{C}}) \lesssim_{\mathcal{I}_{ib}} \mathcal{O}(\llbracket N \rrbracket_{\mathcal{C}}) . & (ii) \end{aligned}$$

Proof. (i) (\Leftarrow) This is the Inequational Soundness result (Theorem 6.5.1). (\Rightarrow) We follow the same argument as the proof of Full Abstraction of PCF [AM97]. Suppose that $\llbracket M \rrbracket_{\mathcal{C}} \lesssim_{\mathcal{I}_{ib}} \llbracket N \rrbracket_{\mathcal{C}}$. Then by definition of the preorder $\lesssim_{\mathcal{I}_{ib}}$, there exists a P-i.j. strategy $\alpha : (\Gamma \rightarrow \llbracket T \rrbracket) \rightarrow \mathbf{exp}$ such that $\llbracket M \rrbracket \circ \alpha = \top$ and $\llbracket N \rrbracket \circ \alpha = \perp$. α can be chosen to be compact. Moreover since $\text{ord}(T) \geq \text{ord}(\mathbf{exp}) = 0$, the strategy α is closed P-i.j. By the definability result for safe PCF (Prop. 5.7.1), there exists a closed safe term-in-context $\vdash_s \lambda z^{\Gamma \rightarrow T}.Q : (\Gamma \rightarrow T) \rightarrow \mathbf{exp}$ such that $\llbracket \lambda z^{\Gamma \rightarrow T}.Q \rrbracket = \alpha$. Using the application rule and the abstraction we can then form the safe program context: $- : T \vdash_s (\lambda z^{\Gamma \rightarrow T}.Q)(\lambda y^{\Gamma}.-) \equiv C[-] : \mathbf{exp}$. In particular, the subterm $\lambda y^{\Gamma}.-$ is safe because we have $\text{ord} - = \text{ord} T > \text{ord} \Gamma$ by assumption. Clearly, $\llbracket C[-] \rrbracket \cong \llbracket \lambda z^T.Q \rrbracket = \alpha$ therefore by Computational Adequacy it follows that $C[M] \Downarrow$ and $C[M] \not\Downarrow$.

(ii) In the definition of the preorder $\lesssim_{\mathcal{I}_{ib}}$, the test strategy α ranges over P-i.j. strategies therefore by Lemma 6.4.1 α can only interact with O-i.j. plays. Hence for any strategy σ in \mathcal{C} , $\mathcal{O}(\sigma)$ and σ are in the same $\lesssim_{\mathcal{I}_{ib}}$ -equivalence class. \square

Full abstraction of safe PCF

Although the small-step operational semantics of PCF and safe PCF differ—the former contracts β -redexes one at a time whereas the latter contracts “consecutive” β -redexes in a single step—they have the same big-step operational semantics: a safe term evaluates to a value in safe PCF if and only if it evaluates to the same value in PCF. Hence the operational semantics of safe PCF is given by the same relation \Downarrow as PCF.

We now consider the restrictions of the relations \sqsubseteq and \sqsubseteq_s on $\text{Trm}(\Gamma, T) \times \text{Trm}(\Gamma, T)$ to $\text{Trm}_s(\Gamma, T) \times \text{Trm}_s(\Gamma, T)$. Clearly these restrictions define preorders on $\text{Trm}_s(\Gamma, T)$.

Theorem 6.5.3 (Full abstraction for closed safe PCF terms). *Let M, N be two closed safe PCF terms of the same type. Then*

$$\begin{aligned} M \sqsubseteq_s N &\iff \llbracket M \rrbracket_{\mathcal{I}} \lesssim_{\mathcal{I}_{ib}} \llbracket N \rrbracket_{\mathcal{I}} \\ &\iff \mathcal{O}(\llbracket M \rrbracket_{\mathcal{I}}) \lesssim_{\mathcal{I}_{ib}} \mathcal{O}(\llbracket N \rrbracket_{\mathcal{I}}) . \end{aligned}$$

Proof. Safe closed PCF terms are all in PCF^* therefore the result follows immediately from Theorem 6.5.2 since $\llbracket M \rrbracket_{\mathcal{I}} = \llbracket M \rrbracket_{\mathcal{C}}$ for any safe term M . \square

REMARK 6.5.3 Observe that the condition 6.5 used in Theorem 6.5.2 expresses precisely the negation of the basic property of the safe lambda calculus. Therefore the star fragment of safe PCF is precisely given by the set of *closed* safe terms. That is why our full abstraction result holds only for the fragment of PCF consisting of *closed* terms.

Full abstraction fails for open terms. For instance the family of opened safe terms $\text{cond } 0 \ x \ i$ for $i \in \mathbb{N}$ are all in the same \sqsubseteq_s -equivalence class although their denotations are not in the same $\lesssim_{\mathcal{I}_{ib}}$ -equivalence class.

In fact the observational relation \sqsubseteq_s trivially equates all open safe terms of a given type! This is due to the fact that for any open safe term M , there is no safe context $C[-]$ such that the term $C[M]$ is closed. (See remark 6.5.2).

Full abstraction of Safe Idealized Algol

The proof of full abstraction of Idealized Algol is based on the Innocent Factorization theorem:

Theorem 6.5.4 (Innocent Factorization [AM97]). *For any strategy σ on an a IA game A , there exists an innocent strategy $\tau : !\text{var} \multimap A$ such that $\sigma = \text{cell}; \tau$.*

A version of this theorem also holds for safe IA:

Lemma 6.5.4. *For any closed P-i.j. strategy σ on an a IA game A , there is an innocent strategy $\mu : !\text{var} \multimap A$ which is closed P-i.j. modulo $\text{In}(!\text{var})$ and such that $\sigma = \text{cell}; \mu$.*

Proof. By the Factorization Theorem we have that $\sigma = \text{cell}; \tau$ for some innocent strategy $\tau : !\text{var} \multimap A$. Observe that τ is not necessarily P-i.j. modulo $\text{In}(\llbracket !\text{var} \rrbracket)$, although σ is P-i.j. (See the following remark.) However all the plays of τ interacting with cell are P-i.j. modulo $\text{In}(\llbracket !\text{var} \rrbracket)$. Indeed, take an interaction play $u \in \text{Int}(I, !\text{var}, A)$ ending with an A -move. It is easy to see that P-view of the play $u \upharpoonright I, A$ is obtained from the P-view of the play $u \upharpoonright !\text{var}, A$ by removing the moves played in $\llbracket !\text{var} \rrbracket$. Thus because the question moves of the game $\llbracket !\text{var} \rrbracket$ are of order 0, since $u \upharpoonright I, A$ is P-i.j., so must be $u \upharpoonright !\text{var}, A$.

Take μ to be the strategy obtained by truncating all the plays in τ that are not P-i.j. Then clearly μ is P-i.j. modulo $\text{In}(\llbracket !\text{var} \rrbracket)$ and verifies $\sigma = \text{cell}; \mu$. \square

REMARK 6.5.4 In the previous proof, we mentioned that it is possible for $\text{cell}; \tau$ to be P-i.j. even when τ is not P-i.j. modulo $\text{In}(\llbracket !\text{var} \rrbracket)$. Here is an example illustrating this fact. The IA term

$$\begin{aligned} x : \text{var} \vdash \lambda f^2 y^{\text{exp}}. \text{seq}(\text{assign } x \ 0) (\text{cond}(\text{deref } x) \ 0 \ (f(\lambda z^{\text{exp}}. \underline{y}))) &\equiv M \\ : \text{var}^0 \rightarrow ((\text{exp}^1 \rightarrow \text{exp}^2) \rightarrow \text{exp}^3) \rightarrow \text{exp}^4 \rightarrow \text{exp}^5 \end{aligned}$$

is unsafe because it contains the unsafe occurrence y , and its denotation is not P-i.j. modulo $\text{In}(\llbracket !\text{var} \rrbracket)$ because it contains the play:



The term **new** x in M , however, is observationally equivalent to 0 and therefore its denotation is P-i.j.

As in the IA case, the factorization result can be used to show that all the compact closed P-i.j. strategies on IA types are definable in safe IA. Inequational Soundness of the game model follows from that of IA. We then obtain:

Theorem 6.5.5 (Full abstraction for closed safe IA terms). *Let $\vdash_s M : T$ and $\vdash_s N : T$ be two safe closed IA terms. Then:*

$$\begin{aligned} M \sqsubseteq_s N &\iff \llbracket M \rrbracket_{\mathcal{I}} \lesssim_{\mathcal{I}_b} \llbracket N \rrbracket_{\mathcal{I}} \\ &\iff \mathcal{O}(\llbracket M \rrbracket_{\mathcal{I}}) \lesssim_{\mathcal{I}_b} \mathcal{O}(\llbracket N \rrbracket_{\mathcal{I}}) . \end{aligned}$$

where the preorder \sqsubseteq_s is defined similarly as for safe PCF.

Proof. This result follows from the definability result as in the case of safe PCF. \square

6.6 Algorithmic game semantics

The game model of safe IA is greatly simplified since justification pointers are unnecessary. By the Characterization Theorem (Sec. 2.3.7), \cong -observational equivalence of IA terms is characterized by equality of the set of complete plays. Thus for safe terms, \cong -equivalence is characterized by equality of the set of underlying sequence of moves without justification pointers. This simplification suggests applications in algorithmic game semantics.

We show here that up to order 3, IA is a conservative extension of safe IA. This means that the observational equivalence relations \cong_s and \cong coincide. Therefore, all the upper-bounds on the complexity of observational equivalence that are known for the order-3 fragments of IA also hold for safe IA. We then show that the Characterization Theorem also holds for observational equivalence of safe IA with respect to safe context: two terms are \cong_s -equivalent if the sets of complete plays of their denotation are the same. Consequently, we can show that up to order 3, the complexity lower-bounds that are already known for IA also hold in safe IA.

Observational equivalence

Proposition 6.6.1.

- (i) Up to order 3, it is conservative, with respect to observational equivalence, to add unsafe context to safe ones. Formally for any closed IA terms M, N we have:

$$M \sqsubseteq_s N \iff M \sqsubseteq N .$$

- (ii) Adding unsafe context is not conservative at order 4 for Idealized Algol.

Proof. (i) Let A be an order-3 type and M, N be two IA terms of type A .

$$\begin{aligned} M \sqsubseteq N &\iff \llbracket M \rrbracket \lesssim \llbracket N \rrbracket && \text{by Full abstraction of IA.} \\ &\iff \llbracket M \rrbracket \lesssim_{\mathcal{I}} \llbracket N \rrbracket && \text{Corollary 6.4.1} \\ &\iff M \sqsubseteq_s N && \text{by full abstraction of IA w.r.t. safe contexts.} \end{aligned}$$

(ii) The idea is to start from some term E and construct a term D that behaves like E except that it has a “hidden” behaviour which can only be triggered when the Opponent plays in some particular way that is not incrementally justified. Take the following order-4 IA terms:

$$\begin{aligned} E &\equiv \lambda\varphi^{(2,2,0)}.\varphi(\lambda u_1^o.u_1 \text{ skip})(\lambda u_2^o.u_2 \text{ skip}) : ((2, 2, 0), 0) \\ D &\equiv \lambda\varphi^{(2,2,0)}.\text{new } LAST := 0 \text{ in} \\ &\quad \varphi(\lambda u_1^o.\text{new } PREV := !LAST \text{ in } LAST := 1; u_1(!LAST = 1); LAST := PREV) \\ &\quad (\lambda u_2^o.\text{new } PREV := !LAST \text{ in } LAST := 2; u_2(!LAST = 2); LAST := PREV) \\ &\quad : ((2, 2, 0), 0) \end{aligned}$$

where we use the type abbreviations 0 for **com** and $k + 1 = k \rightarrow \text{com}$ for $k \geq 0$, and for every term $T : \mathbf{exp}$, the assertion operator $[T]$ is syntactic sugar for **cond** $T \ \Omega \ \text{skip}$ (i.e., the term that does nothing if T evaluates to a positive number and goes into an infinite loop otherwise).

The two terms M and N are not observationally equivalent in PCF because the unsafe context

$$C[-] = -(\lambda w_1^2 w_2^2.w_1(\lambda x^o.w_2(\lambda y^o.\underline{x})))$$

can separate them: we have $C[D] \not\Downarrow$ and $C[E] \Downarrow$. In safe PCF, however, these two terms are observationally equivalent: Let $C[-]$ be a safe context. We claim that when evaluating $C[D]$, the variable $LAST$ always contains the index of the last called φ 's parameter and therefore the assertion tests in D always succeed. This can be shown by induction on the length of the interaction play between $\llbracket C[-] \rrbracket$ and $\llbracket D \rrbracket$. We give here an informal argument. Assume that the context makes a single call to D . (The argument can be easily adapted to the general case.) During the evaluation, whenever a parameter of φ is called, D first sets the variable $LAST$ to the parameter index i and then calls the Opponent's parameter u_i . At that point, O can either make another call to one of φ 's parameter, or it can call the parameter of some previous call to some u_j for $j \in \{1, 2\}$. Suppose it does the latter, because it is playing incrementally (since the context is safe) such u_j must necessarily be the u_i that was last called by P . The next step executed by P is then the assertion test which necessarily succeeds because $LAST$ was just set to i . When the call to u_i returns, P restores $LAST$ to the value it originally contained when φ 's parameter was called, thus ensuring that it holds the index of the φ 's parameter that was last called by the context.

Similarly, whenever a call to one of φ 's parameter returns, the Opponent can call the parameter of the *last* (because O plays incrementally) called u_j . Since $LAST$ contains the last called φ 's parameter's index, this again ensures that the assertion test succeeds. \square

Characterization Theorem

We now show that a version of the Characterization Theorem (Sec. 2.3.7) also holds for safe IA:

Theorem 6.6.1 (Characterization Theorem for in \mathcal{I}). *Let σ and τ be two closed P-i.j. strategies on a simple game A in \mathcal{I} . Then*

$$\sigma \lesssim_{\mathcal{I}} \tau \quad \Longleftrightarrow \quad \text{comp}(\mathcal{O}(\sigma)) \subseteq \text{comp}(\mathcal{O}(\tau)) .$$

Proof. By Theorem 6.5.5, $\sigma \lesssim_{\mathcal{I}} \tau$ iff $\mathcal{O}(\sigma) \lesssim_{\mathcal{I}} \mathcal{O}(\tau)$. The rest of the proof then follows the same argument used to prove the original Characterization Theorem for the category \mathcal{C}_b [AM97, Theorem 25], with one subtlety: in the first part of the proof, the fact that $\mathcal{O}(\sigma)$ is O-i.j. guarantees that the strategy $\alpha : A \rightarrow \Sigma$ which “follows the script of s ” is P-incrementally justified. \square

Consequently, observational equivalence of safe IA terms with respect to safe IA contexts is characterized by equality of the set of complete plays.

Classification

Upper bounds By Proposition 6.6.1, all the known upper-bound for IA are also valid for safe IA up to order 3: safe $IA_2 + \text{while}$ is decidable in PSPACE [GM00], $IA_3 + \text{while}$ is decidable in EXPTIME for terms in β -nf [MW05], and $IA_3 + Y_0$ is decidable with a complexity that is at most doubly exponentially larger than that of the DPDA equivalence problem [MOW05].

Lower bounds

Theorem 6.6.2 (Ong [Ong02]). *Observational equivalence of $IA_2 + Y_1$ is undecidable.*

The proof of this theorem proceeds by reduction of the QUEUE-HALTING problem to the observational equivalence of two $IA_2 + Y_1$ programs: given a QUEUE program P , a $IA_2 + Y_1$ term $\vdash M_P : \text{com}$ is defined such that M_P simulates P in the sense that P terminates if and only if M_P is equivalent to **skip**. It turns out that the encoding term M_P [Ong02] is safe therefore:

Corollary 6.6.3. *Observational equivalence of safe $IA_2 + Y_1$ is undecidable.*

For $IA_3 + \text{while}$, it was shown that the Containment Problem for DPDA can be reduced to observational approximation in $IA_1 + Y_0$ [MOW05, Proposition 1]. Therefore observational approximation is undecidable for $IA_1 + Y_0$ terms, and observational equivalence is at least as hard as DPDA Equivalence.

Corollary 6.6.4. *For safe $IA_2 + Y_0$, observational approximation is undecidable and observational equivalence is at least as hard as DPDA Equivalence.*

Proof. The original encoding [MOW05] is not safe because it contains an occurrence of a variable $x : \text{exp}$ occurring in the body of a μ -abstraction μz with $\text{ord } z = \text{ord } x$. An equivalent safe encoding can be obtained by replacing the free variable $x : \text{exp}$ by a variable of type $\text{exp} \rightarrow \text{exp}$, thus giving an encoding in safe $IA_2 + Y_0$.

Let \mathcal{B} be a DPDA over an alphabet Σ . We write $N(\mathcal{B})$ to denote the language accepted by \mathcal{B} . We identify values of type exp with $\Sigma \cup \{0\}$ and we consider the game $G = (\text{exp}^2 \rightarrow \text{exp}^1) \rightarrow \text{com}$ whose set of moves is given by $M_G = \{q^1, q^2\} \cup \Sigma \cup \{\text{run}, \text{done}\}$. Following [MOW05], for any language $L \subseteq \Sigma^*$, we define $\hat{L} \subseteq M_G^*$ as $\hat{L} = \{\text{run } q^1 q^2 0 x_1 \cdots q^1 q^2 0 x_n \text{ done} \mid x_1 \dots x_n \in L\}$. We have $\hat{L}_1 = \hat{L}_2$ iff $L_1 = L_2$.

Claim: There exists a safe term $z : \text{exp}^2 \rightarrow \text{exp}^1 \vdash Q_{\mathcal{B}} : \text{com}$ such that the set of underlying sequence of moves of the complete plays of $\llbracket z : \text{exp}^2 \rightarrow \text{exp}^1 \vdash Q_{\mathcal{B}} : \text{com} \rrbracket$ is equal to $\hat{N}(\mathcal{B})$. This term $Q_{\mathcal{B}}$ is obtained from the term $M_{\mathcal{B}}$ used in the original encoding, by replacing the free variable $x : \text{exp}$ in $M_{\mathcal{B}}$ by a variable z of type $\text{exp} \rightarrow \text{exp}$ and by replacing the subterm “ $CH := x$ ” by “ $CH := z 0$ ”. We can then conclude as in the proof for $IA_1 + Y_0$ [MOW05]. \square

For $IA_3 + \text{while}$, Murawski et al. showed that observational equivalence is EXPTIME-hard by a reduction from the equivalence problem of nondeterministic automata on binary trees [MW05, Corollary 2]. The encoding used in the paper is unsafe but it can be easily changed into an equivalent safe term of the same order using the same trick as in the previous proof. (The variable $y : \text{exp}$ is replaced by $y : \text{exp} \rightarrow \text{exp}$ and “ $Z := y$ ” is replaced by “ $Z := y\ 0$ ”). Hence:

Proposition 6.6.2. *Observational equivalence in safe $IA_3 + \text{while}$ is EXPTIME-hard.*

At order 4, since adding unsafe context is not conservative (Prop. 6.6.1) we need to distinct two problems: deciding \cong -equivalence and deciding \cong_s -equivalence (*i.e.*, observational equivalence defined with respect to safe context only).

Murawski showed that \cong -observational equivalence is undecidable at order 4 [Mur03]. He considers Γ -machines, a Turing complete class of devices, and show that for any such machine, there is an IA_4 -term M such that the machine accept the empty string if and only if the set of complete plays of $\llbracket M \rrbracket$ is not empty. This show that \cong -observational equivalence is undecidable. It turns out that Murawski’s encoding is safe, therefore:

Corollary 6.6.5. *\cong -observational equivalence for safe IA_4 is undecidable.*

The fact that contexts are not restricted to be safe is crucial in this simulation. The ADD operation of Γ -machines is for instance simulated using plays that are not O-i.j.² Thus the same argument can be used to show undecidability of \cong_s -observational equivalence. We make the following conjecture:

Conjecture 6.6.6. *\cong_s -observational equivalence for safe IA_4 is decidable.*

The idea is that by the Characterization Theorem for safe IA (Theorem 6.6.1), two terms are equivalent iff the sets of complete O-incrementally justified plays of their denotation are equal. But for such plays, all the pointers can be uniquely recovered from the underlying sequence of moves. Therefore observational equivalence is characterized by equality of the sequences of moves underlying the sequence of complete O-i.j. plays. I believe that at order 4, such sequences can be represented by a DPDA. This would imply the above conjecture.

All the previous results are recapitulated in Table 6.6.

L	Obs. eq.	Finitary fragments				
		order 2 + while	order 2 + Y_1	order 3 + while	order 3 + Y_0	order 4
IA	\cong	PSPACE ⁽¹⁾ \preceq DFA	U ⁽²⁾	EXPTIME-hard ⁽³⁾ EXPTIME-complete for β -nf \preceq VPA	D ⁽⁴⁾ \preceq_{exp} DPDA \succcurlyeq DPDA	U ⁽⁵⁾
	\cong_s					? ⁽⁶⁾
Safe IA	\cong					U
	\cong_s					?

U = Undecidable

D = Decidable with unknown complexity

$\preceq P$ = “reducible to problem P ”

$\succcurlyeq P$ = “at least as hard as problem P ”

(1) [GM00] (2) [Ong02] (3) [MW05] (4) [MOW05] (5) [Mur03] (6) The Characterization Theorem does not hold in that case.

Table 6.2: Complexity of observational equivalence for finitary fragments of safe IA.

²In the paper, the plays ending with the move r_4 are not O-i.j.

Fragment	Representable languages	Machine equivalent
IA_0	singleton sets + empty set	–
IA_1	finite languages with the prefix property	–
IA_2	regular languages	Finite State Automata
IA_3	context free languages	Pushdown Automata
IA_4	recursively enumerable	Turing Machines

Table 6.3: Murawski representability.

Expressivity of safe IA

Murawski introduced a notion of representability of languages by IA terms [Mur03]: a language is represented by (some erasure homomorphism of) the set of complete plays of the term. He shows that the class of languages representable by second-order terms is precisely the regular languages; for third-order terms it is the class of context-free languages; and for terms of order 4 and above, it is the full class of recursively enumerable languages. These results are recapitulated in Table 6.6.

What are the representable languages in the safe fragments of IA? It turns out that up to order 3, the safety constraint does not alter expressivity:

Proposition 6.6.3. *For $0 \leq k \leq 3$, safe IA_k and IA_k are equi-expressive (in terms of Murawski-representable language).*

Proof. Unsafety only appears at order 3 therefore the same languages are representable in IA_i and safe IA_i for $i < 3$. The order-3 term used by Murawski’s encoding [Mur03] to represent context-free languages is unsafe, but it can be easily turned into a safe term by replacing the variable $c : \mathbf{exp}$ by a variable of type $(\mathbf{com} \rightarrow \mathbf{com}) \rightarrow \mathbf{exp}$ and changing the code “ $INPUT := c$ ” into “ $INPUT := c(\lambda z.z)$ ”. \square

It is not known which languages are expressible in higher-order fragments of safe IA. Recall that regular languages are the languages definable by 0-DPDAs, and context-free languages are those definable by DPDAs, so a possible conjecture is: “Murawski-representable in safe IA_n for $n \geq 2$ are the $(n - 2)$ -DPDA definable word languages”. It is not clear, however, how to interpret the higher-order “push” DPDA instructions in terms of game semantic moves. If such result were to be proved then the question of decidability of higher-order DPDA would become relevant to the observational equivalence problem: the undecidability of the former would imply that of the latter.

Chapter 7

Conclusion

7.1 Summary of contribution

Safety is a syntactic constraint for higher-order grammars. A grammar is *safe* if the right-hand side of each rule is such that no subterm occurring in operand position contains parameters of order smaller than the order of the subterm. Motivated by the appealing algorithmic properties of safety, we derived a new typing system, the *safe lambda calculus*, by imposing this syntactic constraint on the simply typed lambda calculus. The salient property of this calculus is that it is not necessary to rename variables when performing substitution. Thus in some sense, safe terms are “easier” to compute than unsafe ones. Computation in our calculus is standardly done via the concept of β -reduction. Safety is not preserved by beta-reduction in general, but it is preserved when sufficiently many consecutive redexes are contracted simultaneously. This is formalized by the notion of *safe beta-reduction*: If a safe term contains a β -redex then this redex can always be “enlarged” into a group of consecutive beta-redexes, called a safe redex, such that contracting all of them produces a safe term. The notion of normal form thus remains unchanged. Further, safety is an extensional property: a term is safe if and only if its eta-long normal form is.

The typing system of the safe lambda calculus has desirable properties: the type-checking (Can a given type be assigned to a given term?) and typability (Given a term, is there a type that can be assigned to it?) problems are both decidable. On the other-hand, we only know that the type-inhabitation problem (Given a type, is there a safe term of that type?) is at least semi-decidable (there is an algorithm that tells if a type is inhabited by a safe term in a finite amount of time if it is the case, but may not terminate otherwise).

The loss of expressivity incurred by safety can be characterized in terms of expressible numeric functions: they are precisely the multivariate polynomials; the conditional operator, which is definable in the lambda calculus, is not expressible by any safe term. In terms of representable word functions, these are given by the set containing the projections, constant functions, concatenation and substitution and closed by composition.

We then looked at the complexity of the calculus by considering the beta-equivalence problem: we hinted that it probably lies in the complexity class ELEMENTARY by showing how both Statman and Mairson’s encoding of finite type theory in the simply typed lambda calculus fail in the safe fragment. We showed however that the problem is PSPACE-hard.

Seeking for a semantic explanation of the safety constraint, we focused on the analysis of the game semantics of safe terms. This led us to the other main contribution of this thesis: the development of a new presentation of Game semantics based on the theory of traversals [Ong06a]. Essentially, traversals implement a version of β -reduction in which beta-redex are computed locally as opposed to a global approach based on substitution. The soundness of the traversal theory as a model of computation is ensured by the correspondence with game semantics: traversals are just uncovering of plays in game semantics.

Armed with the Correspondence Theorem, we were able to give a precise account of the game

semantics of the safe lambda-calculus. A notable property of safe terms is that its variables are incrementally-bound: the binder of a variable node x in the computation tree is precisely the last lambda-node in the path from x to the root with order strictly greater than $\text{ord } x$. By the Correspondence Theorem, this implies that the strategy denotation of a safe term is *P-incrementally justified*. In such strategy, a P-question's justifier is given by the last O-move in the P-view with greater order.

In the last chapter we finally investigated the categorical model of the safe lambda calculus. We proposed the notion of Incremental Closed Category (ICC) that soundly interprets the safe lambda calculus in the same way Cartesian Closed Categories model the simply typed lambda calculus. We then exhibited such an ICC by constructing a game model of P-incrementally justified strategies. (We showed in particular that P-incremental justified strategies compose.)

To conclude, we looked at safety from the point of view of *Algorithmic Game Semantics*. We considered the problem of observational equivalence of IA term with respect to *safe* contexts. By suitably constraining O-moves by the dual notion of *O-incremental justification*, we obtain a model of safe PCF and safe IA that is fully abstract with respect to this notion of observational equivalence. Furthermore, the model is effectively presentable: two safe terms are observationally equivalent with respect to safe context if and only if their denotations have the same set of *complete O-incrementally justified* plays.

Up to order 3, the addition of unsafe contexts to safe ones is conservative with respect to observational equivalence. Furthermore, all the complexity results—lower and upper bounds—known about observational equivalence of the (unrestricted) lower-order fragments of IA also hold in the safe sub-fragments. At order-4, however, the notion of observational equivalence with respect to unrestricted contexts differs from the one defined with respect to safe contexts only. Concerning the latter, we conjecture that the restriction of the problem to *safe* terms (*i.e.*, safe observational equivalence of safe IA_4 terms) is reducible to the DPDA-equivalence problem (which is decidable).

7.2 Further works

The nature of the safe lambda calculus is still not completely understood. Some questions remain about the safe lambda-calculus pertaining for instance to its computational power, the complexity classes that it characterizes and its interpretation under the Curry-Howard isomorphism. We now propose possible directions for further works and highlight some open questions.

Type theory

One of the most pressing question concerns the complexity of the safe lambda calculus. We have shown that the beta-equivalence problem is PSPACE-hard, but this lower-bound may be very coarse. Further investigations need to be done to determine an upper-bound.

Another open problem is the question of decidability of type inhabitation. At the moment we already know that it is semi-decidable: there is an algorithm that, given a simple type, can exhibit a safe inhabitant if it exists but may not terminate otherwise.

Extensions

We have defined a notion of safety for simply typed terms (and also for untyped terms by means of a Curry-like version of the typing system). Is there any generalization to more complicated typing system such as the second-order lambda calculus?

Logic

What kind of reasoning principles does the safe lambda calculus support via the Curry-Howard Isomorphism? How expressive is the safe fragment of intuitionistic implication logic? Is the logic decidable?—or equivalently is type inhabitation decidable in the safe lambda calculus?

Computational complexity

Can the safe lambda calculus help to characterize complexity classes? There are already many such results in the unrestricted case: Leivant and Marion [LM93] considered for instance an “unpure” variation of the simply typed lambda calculus extended with constructors, destructors and conditionals, and obtain several characterizations of the polytime-computable numeric functions in that language.

Hillebrand, Kanellakis and Mairson [HKM96] considered the problem from a database point of view. Instead of encoding numeric functions, they looked at the database queries that are encodable in the simply typed lambda calculus and gave a precise characterization of PTIME: the polynomial time queries are those expressible in the 4th order fragment of the simply typed lambda calculus. This result was later extended to give characterizations of the standard complexity classes PSPACE, k -EXPTIME, k -EXPSPACE ($k \geq 1$) and ELEMENTARY at higher-orders [HK96].

More research needs to be done to see if similar characterizations can be obtained in the safe lambda calculus.

Expressibility

Functions over free algebras

What are the functions over free-algebras definable in the safe simply typed lambda calculus?

There is an isomorphism between binary trees and closed simply typed terms of type $\tau = (o \rightarrow o \rightarrow o) \rightarrow o \rightarrow o$. Thus any closed term of type $\tau \rightarrow \tau \rightarrow \dots \rightarrow \tau$ represents an n -ary function over trees. Zaionc [Zai88] and Leivant [Lei93] gave a characterization of the set of tree functions representable in the simply typed lambda calculus: it is precisely the minimal set containing constant functions, projections and closed under composition and limited primitive recursion. Zaionc showed that the same characterization holds for the general case of functions expressed over free algebras [Zai91]: they are given by the minimal set containing constant functions, projections and closed under composition and limited primitive recursion. This result subsumes Schwichtenberg’s result on definable numeric functions as well as Zaionc’s own results on definable word and tree functions.

All these basic operations are safe except limited primitive recursion. This suggests that one needs to restrict further the primitive recursion in order to obtain a characterization of free-algebra functions representable in the safe lambda calculus. Such result would generalize our expressivity result for numeric and word functions from Sec. 3.3.

Murawski-expressibility

Murawski introduced a notion of language expressibility by game semantics [Mur03]. He showed that the 4th order finitary fragment of IA is expressive enough to give the full class of recursively enumerable languages. Does the safe fragment have the same expressive power? Another line of research would be to investigate whether the class of word languages recognizable by higher-order pushdown automata can be characterized in Murawski’s sense by some higher-order fragment of safe IA.

Trees and word languages

The impact of safety on the expressivity of higher-order recursion schemes was studied in de Miranda’s thesis [dM06]. At order 2 and for word languages, safety is not a genuine constraint if we allow non-determinism [AdMO05b]; de Miranda and Urzyczyn conjectured that for *deterministic* higher-order grammars, safety is a proper restriction. Urzyczyn even proposed an unsafe deterministic higher-order recursion scheme generating a word language that he conjectured to be inherently unsafe (*i.e.*, that cannot be generated by any deterministic safe grammar). At the time of this writing, though, this remains a conjecture. The traversal theory seems to be a promising tool to investigate the problem.

Game semantics

Is the game model of safe PCF universal? (*i.e.*, is every recursive incremental strategy denoted by some safe PCF term?) Is there a category of O-incrementally justified strategies?

Compilation of safe recursion schemes to pushdown automata

We have mentioned before the equi-expressivity result about safe homogenously-typed higher-order recursion schemes and higher-order pushdown automata: these two devices generate the same class of infinite trees. Hague et al. generalized this result to unrestricted recursion scheme; one direction relies on the traversal theory: an order n recursion scheme can be compiled into an equivalent order n *collapsible* pushdown automaton which proceeds by computing the set of traversals of the recursion scheme's computation graph [HMOS08]. We conjecture that when the safety constraint is imposed, this encoding can be specialized into a higher-order pushdown automaton (without the collapse operation). Such result would give an alternative proof of Knapik et al.'s equi-expressivity result [KNU02].

Algorithmic game semantics

Is observational equivalence for safe IA_4 decidable? We have seen that up to order 3, the problem of observational equivalence has the same complexity in the safe finitary fragments as in the unrestricted finitary fragments. At order 4 the picture remains unclear. Murawski [Mur03] showed the undecidability of program equivalence in IA_i for $i \geq 4$ by encoding Turing machine computations using finitary IA_4 terms. Because his encoding relies on unsafe terms, the argument cannot be transposed to the safe fragment of IA. The question of whether observational equivalence of safe IA_4 is decidable thus remains open.

PUR languages

In this thesis, we have shown that the safety constrained produces languages whose game semantics enjoy the property that some justification pointers are uniquely recoverable from the underlying sequence of moves. Safe IA_3 is an example of language in which *all* pointers are recoverable. We name this class *PUR* for “*Pointer Uniquely Recoverable*”. Finitary IA_2 (finite base types and no recursion) is the paradigmatic example of a PUR-language (The fact that it is a sublanguage of Safe IA_3 is another proof of this fact). But safe fragments are clearly not the only PUR-languages: singleton languages (*i.e.*, containing only one term) are trivial examples of PUR languages. Also the language consisting of all IA_3 terms whose beta-reduction is safe is also a PUR language.

A more interesting example is *Serially Re-entrant Idealized Algol* [Abr01b], a version of IA where multiple uses of arguments are allowed only if they do not “overlap in time”. In the game semantics denotation of a SRIA term there is at most one pending occurrence of a question at any time. Each move has therefore a unique justifier and consequently justification pointers may be ignored. Safe IA is not a sublanguage of SRIA. One reason for this is that none of the two Kierstead terms $\lambda f.f(\lambda x.f(\lambda y.y))$ and $\lambda f.f(\lambda x.f(\lambda y.x))$ are Serially Re-entrant whereas the first one is safe. Conversely, SRIA is not a sublanguage of safe IA since the term $\lambda fg.f(\lambda x.g(\lambda y.x))$ where $f, g : ((o, o), o)$ belongs to SRIA but not to safe IA.

Another way to generate PUR-languages may consist in constraining types. Joly introduced a notion of “complexity” for lambda-terms and proved that there is a constant bounding the complexity of every closed normal lambda-term of a given type T if and only if T can be generated from a finite set of combinators; Consequently, the only inhabited finitely generated types are the types of order ≤ 2 and the types $(A_1, A_2, \dots, A_n, o)$ such that for all $i = 1..n$: $A_i = o$, $A_i = o \rightarrow o$ or $A_i = (o^k \rightarrow o) \rightarrow o$ [Jol01]. We already know that imposing the first type restriction to Finitary IA leads to a PUR language. Does the second restriction also give a PUR language?

With a view to algorithmic game semantics and its applications, the PUR class is of particular interest. Indeed, PUR-languages are good candidates of languages with decidable observational

equivalence. This is because the simplification of the game semantic model resulting from the nonnecessity of pointers makes the observational equivalence problem more manageable: In IA, for instance one just need to compare the set of complete plays underlying the denotation of a term, forgetting the justification pointers altogether. For lower-order fragments, a machine characterization of this set is sometimes possible (*e.g.*, finite-state automaton at order-2, and deterministic pushdown automata for the order-3 fragment with Y_0 recursion), subsequently leading to decidability and complexity results for the observational equivalence problem.

Bibliography

- [Abr01a] S. ABRAMSKY – Algorithmic game semantics: a tutorial introduction. In *Proceedings of 2001 Marktoberdorf International Summer School*, 2001.
- [Abr01b] —, Semantics via game theory. In *Marktoberdorf International Summer School*, 2001, Lecture slides.
- [ADLR94] A. ASPERTI, V. DANOS, C. LANEVE and L. REGNIER – Paths in the lambda-calculus. In *LICS*, IEEE Computer Society, 1994, p. 426–436.
- [AdMO04] K. AEHLIG, J. G. DE MIRANDA and C.-H. L. ONG – Safety is not a restriction at level 2 for string languages. Tech. report, University of Oxford, 2004.
- [AdMO05a] —, The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA* (P. Urzyczyn, ed.), Lecture Notes in Computer Science, vol. 3461, Springer, 2005, p. 39–54.
- [AdMO05b] —, Safety is not a restriction at level 2 for string languages. In *FoSSaCS* [Sas05], p. 490–504.
- [AGOM03] S. ABRAMSKY, D. R. GHICA, C.-H. L. ONG and A. MURAWSKI – Algorithmic Game Semantics and Component-Based Verification. In *Proceedings of SAVBCS 2003: Specification and Verification of Component-Based Systems, Workshop at ES-EC/FASE 2003*, 2003, published as Technical Report 03-11, Department of Computer Science, Iowa State University, p. 66–74.
- [AHM98] S. ABRAMSKY, K. HONDA and G. MCCUSKER – A fully abstract game semantics for general references. In *LICS*, 1998, p. 334–344.
- [Aho68] A. V. AHO – Indexed grammars – an extension of context-free grammars. *J. ACM* **15** (1968), no. 4, p. 647–671.
- [AJ92] S. ABRAMSKY and R. JAGADEESAN – Games and full completeness for multiplicative linear logic. In *Foundations of Software Technology and Theoretical Computer Science (FST-TCS'92)* (New Delhi, India) (R. Shyamasundar, ed.), 1992, p. 291–301.
- [AJ05] —, A game semantics for generic polymorphism. *Ann. Pure Appl. Logic* **133** (2005), no. 1-3, p. 3–37.
- [AM97] S. ABRAMSKY and G. MCCUSKER – Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In *Algol-like languages* (P. W. O’Hearn and R. D. Tennent, ed.), Birkhäuser, 1997.
- [AM98a] —, Call-by-value games. In *Computer Science Logic: 11th International Workshop Proceedings* (M. Nielsen and W. Thomas, ed.), Springer-Verlag, 1998.
- [AM98b] —, Game semantics. In *Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School* (H. Schwichtenberg and U. Berger, ed.), Springer-Verlag, 1998, Lecture notes, p. 1–56.

- [AM99] —, Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science* **227** (1999), no. 1–2, p. 3–42.
- [AMJ94] S. ABRAMSKY, P. MALACARIA and R. JAGADEESAN – Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, 1994, p. 1–15.
- [Asp] A. ASPERTI – $P = NP$, up to sharing.
- [Bar84] H. P. BARENDREGT – *The Lambda Calculus – Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, 1984.
- [Bar92] H. BARENDREGT – Lambda calculi with types. In *Handbook of Logic in Computer Science* (S. Abramsky, D. M. Gabbay and T. Maibaum, ed.), vol. 2, Clarendon Press, 1992, p. 117–309.
- [Ber78] G. BERRY – Stable models of typed lambda-calculi. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming* (London, UK), Springer-Verlag, 1978, p. 72–89.
- [Ber79] —, Modèles complément adéquats et stable des lambda-calculs typés. Phd thesis, Université Paris VII, 1979.
- [Blu08] W. BLUM – A tool for constructing structures generated by higher-order recursion schemes and collapsible pushdown automata. <http://william.famille-blum.org/research/tools.html>, 2008.
- [BO07] W. BLUM and C.-H. L. ONG – The safe lambda calculus. In *TLCA* (S. R. D. Rocca, ed.), Lecture Notes in Computer Science, vol. 4583, Springer, 2007, p. 39–53.
- [Cau02] D. CAUCAL – On infinite terms having a decidable monadic theory. *Lecture Notes in Computer Science* **2420** (2002), p. 165–176.
- [CM64] J. COCKE and M. MINSKY – Universality of tag systems with $p = 2$. *J. ACM* **11** (1964), no. 1, p. 15–20.
- [Cro93] R. CROLE – *Categories for types*. Cambridge Mathematical Textbooks, Cambridge University Press, 1993.
- [Dam82] W. DAMM – The IO- and OI-hierarchy. *TCS* **20** (1982), p. 95–207.
- [DG86] W. DAMM and A. GOERDT – An automata-theoretical characterization of the OI-hierarchy. *Information and Control* **71** (1986), no. 1–2, p. 1–32.
- [DGL05] A. DIMOVSKI, D. R. GHICA and R. LAZIC – Data-abstraction refinement: A game semantic approach. In *SAS* (C. Hankin and I. Siveroni, ed.), Lecture Notes in Computer Science, vol. 3672, Springer, 2005, p. 102–117.
- [DHR96] V. DANOS, H. HERBELIN and L. REGNIER – Game semantics and abstract machines. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, 27–30 July 1996, p. 394–405.
- [dM06] J. G. DE MIRANDA – Structures generated by higher-order grammars and the safety constraint. D.Phil thesis, University of Oxford, 2006.
- [DR] V. DANOS and L. REGNIER – Head linear reduction.
- [DR93] —, Local and asynchronous beta-reduction (an analysis of girard’s execution formula). In *Proceedings of the Eighth Annual IEEE Symp. on Logic in Computer Science, LICS 1993* (M. Vardi, ed.), IEEE Computer Society Press, June 1993, p. 296–306.

- [FLO83] S. FORTUNE, D. LEIVANT and M. O'DONNELL – The expressiveness of simple and second-order type structures. *J. ACM* **30** (1983), no. 1, p. 151–185.
- [GM00] D. R. GHICA and G. MCCUSKER – Reasoning about idealized ALGOL using regular languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, LNCS, vol. 1853, Springer-Verlag, 2000, p. 103–116.
- [GM03] D. R. GHICA and G. MCCUSKER – The regular-language semantics of second-order Idealized Algol. *Theoretical Computer Science* **309** (2003), no. 1-3, p. 469–502.
- [Gre04] W. GREENLAND – Game semantics for region analysis. Ph.D. thesis, University of Oxford, 2004.
- [Har05] R. HARMER – Innocent game semantics. November 2005, Course notes.
- [Hin97] J. R. HINDLEY – *Basic simple type theory*. Cambridge University Press, New York, NY, USA, 1997.
- [HK96] G. G. HILLEBRAND and P. C. KANELLAKIS – On the expressive power of simply typed and let-polymorphic lambda calculi. In *LICS*, 1996, p. 253–263.
- [HKM96] G. G. HILLEBRAND, P. C. KANELLAKIS and H. G. MAIRSON – Database query languages embedded in the typed lambda calculus. vol. 127, 1996, p. 117–144.
- [HMOS08] M. HAGUE, A. S. MURAWSKI, C.-H. L. ONG and O. SERRE – Collapsible pushdown automata and recursive schemes. *LICS* (2008), p. 452–461.
- [HO93] J. M. E. HYLAND and C.-H. L. ONG – Fair games and full completeness for Multiplicative Linear Logic without the MIX-rule. preprint, 1993.
- [HO00] — , On full abstraction for PCF: I, II, and III. *Information and Computation* **163** (2000), no. 2, p. 285–408.
- [Hoa83] C. A. R. HOARE – Communicating sequential processes. *Commun. ACM* **26** (1983), no. 1, p. 100–106.
- [HS86] J. R. HINDLEY and J. P. SELDIN – *Introduction to combinators and lambda-calculus*. Cambridge University Press, 1986.
- [Hue75] G. P. HUET – A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.* **1** (1975), no. 1, p. 27–57.
- [Hue76] — , Résolution d'équations dans des langages d'ordre 1,2,..., ω . Thèse de doctorat es sciences mathématiques, Université Paris VII, Septembre 1976.
- [HY99] K. HONDA and N. YOSHIDA – Game-theoretic analysis of call-by-value computation. *Theoretical Computer Science* **221** (1999), no. 1-2, p. 393–456.
- [Jol01] T. JOLY – The finitely generated types of the lambda-calculus. In *TLCA*, 2001, p. 240–252.
- [JP76] D. C. JENSEN and T. PIETRZYKOWSKI – Mechanizing *mega*-order type theory through unification. *Theor. Comput. Sci.* **3** (1976), no. 2, p. 123–171.
- [Knu00] D. E. KNUTH – *Fundamental algorithms*. Third ed., The Art of Computer Programming, vol. 1, Addison-Wesley, 2000.
- [KNU02] T. KNAPIK, D. NIWIŃSKI and P. URZYCZYN – Higher-order pushdown trees are easy. In *FOSSACS'02*, Springer, 2002, LNCS Vol. 2303, p. 205–222.

- [Lam86] J. LAMBEK – Cartesian closed categories and typed lambda-calculi. *Proc. of the thirteenth spring school of the LITP on Combinators and functional programming languages table of contents* (1986), p. 136–175.
- [Lam90] J. LAMPING – An algorithm for optimal lambda calculus reduction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA), ACM Press, 1990, p. 16–30.
- [Lei93] D. LEIVANT – Functions over free algebras definable in the simply typed lambda calculus. *Theor. Comput. Sci.* **121** (1993), no. 1&2, p. 309–322.
- [LJBA01] C. S. LEE, N. D. JONES and A. M. BEN-AMRAM – The size-change principle for program termination. In *POPL*, Proceedings ACM Symposium on Principles of Programming Languages, 2001.
- [LM93] D. LEIVANT and J.-Y. MARION – Lambda calculus characterizations of poly-time. In *TLCA* (M. Bezem and J. F. Groote, ed.), Lecture Notes in Computer Science, vol. 664, Springer, 1993, p. 274–288.
- [Loa98a] R. LOADER – Notes on simply typed lambda calculus. February 1998.
- [Loa98b] — , Unary PCF is decidable. *Theoretical Computer Science* **206** (1998), no. 1-2, p. 317–329.
- [Loa01] — , Finitary PCF is not decidable. *Theoretical Computer Science* **266** (2001), no. 1-2, p. 341–364.
- [Lor61] P. LORENZEN – Ein dialogisches konstruktivitätskriterium. In *Infinitistic Methods*. (W. PWN, ed.), 1961, p. 193–200.
- [Mai92] H. G. MAIRSON – A Simple Proof of a Theorem of Statman. *TCS* **103** (1992), no. 2, p. 387–394.
- [Mas74] A. N. MASLOV – The hierarchy of indexed languages of an arbitrary level. *Soviet Math. Dokl.* **15** (1974), p. 1170–1174.
- [Mas76] — , Multilevel stack automata. *Problems of Information Transmission* **12** (1976), p. 38–43.
- [McC96a] G. MCCUSKER – Games and full abstraction for a functional metalanguage with recursive types. Ph.D. thesis, Imperial College, 1996.
- [McC96b] — , Games and full abstraction for FPC. In *Proceedings of the Eleventh Annual IEEE Symp. on Logic in Computer Science, LICS 1996* (E. M. Clarke, ed.), IEEE Computer Society Press, July 1996, p. 174–183.
- [McC03] — , On the semantics of Idealized Algol without the bad-variable constructor. In *Nineteenth Conference on the Mathematical Foundations of Programming Semantics* (ENTCS, ed.), vol. 83, Elsevier, 2003.
- [Mey74] A. R. MEYER – The inherent computational complexity of theories of ordered sets. In *Proc. Int'l. Cong. of Mathematicians*, vol. 2, August 1974, p. 477–482.
- [Min67] M. L. MINSKY – *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
- [MOW05] A. S. MURAWSKI, C.-H. L. ONG and I. WALUKIEWICZ – Idealized algol with ground recursion, and DPDA equivalence. In *ICALP* (L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi and M. Yung, ed.), Lecture Notes in Computer Science, vol. 3580, Springer, 2005, p. 917–929.

- [Mur03] A. S. MURAWSKI – On program equivalence in languages with ground-type references. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, 22-25 June 2003, p. 108–117.
- [Mur05] —, Games for complexity second-order call-by-name programs. *Theoretical Computer Science* **343** (2005), p. 207–236, special issue: Game Theory meets Computer Science, accepted for publication.
- [MW05] A. S. MURAWSKI and I. WALUKIEWICZ – Third-order idealized algol with iteration is decidable. In *FoSSaCS* [Sas05], p. 202–218.
- [Nic94] H. NICKAU – Hereditarily sequential functionals. In *Proc. Symp. Logical Foundations of Computer Science: Logic at St. Petersburg* (A. Nerode and Y. V. Matiyasevich, ed.), Lecture Notes in Computer Science, vol. 813, Springer-Verlag, 1994, p. 253–264.
- [Ong02] C.-H. L. ONG – Observational equivalence of third-order Idealized Algol is decidable. In *Proceedings of IEEE Symposium on Logic in Computer Science, 22-25 July 2002, Copenhagen Denmark*, Computer Society Press, 2002, p. 245–256.
- [Ong04] —, An approach to deciding observational equivalence of algol-like languages. *Ann. Pure Appl. Logic* **130** (2004), no. 1-3, p. 125–171.
- [Ong06a] —, On model-checking trees generated by higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science.*, Computer Society Press, 2006, Extended abstract, p. 81–90.
- [Ong06b] —, On model-checking trees generated by higher-order recursion schemes (technical report). Preprint, 42 pp, 2006.
- [Plo75] G. D. PLOTKIN – Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science* **1** (1975), no. 2, p. 125–159.
- [Plo77] —, LCF considered as a programming language. *Theor. Comput. Sci.* **5** (1977), no. 3, p. 225–255.
- [Rey81] J. C. REYNOLDS – The essence of algol. In *Algorithmic Languages* (J. W. de Bakker and J. C. van Vliet, ed.), IFIP, North-Holland, Amsterdam, 1981, p. 345–372.
- [Sas05] V. SASSONE (ed.) – *Foundations of Software Science and Computational Structures, 8th international conference, FOSSACS 2005, held as part of the joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, proceedings*. Lecture Notes in Computer Science, vol. 3441, Springer, 2005.
- [Sch76] H. SCHWICHTENBERG – Definierbare funktionen im lambda-kalkul mit typen. *Archiv Logik Grundlagenforsch* **17** (1976), p. 113–114.
- [Sch01] A. SCHUBERT – The complexity of beta-reduction in low orders. *Proceedings TLCA 2001* (2001), p. 400–414.
- [Sco69] D. S. SCOTT – A theory of computable function of higher type. Unpublished seminar notes, University of Oxford, 1969.
- [Sco93] —, A type-theoretical alternative to iswim, cuch, owhy. *Theor. Comput. Sci.* **121** (1993), no. 1-2, p. 411–440.
- [Sén01] G. SÉNIZERGUES – $L(A)=L(B)$? decidability results from complete formal systems. *Theor. Comput. Sci.* **251** (2001), no. 1-2, p. 1–166.
- [Ser05] D. SERENI – Simply typed λ -calculus and SCT. Unpublished notes, 2005.

- [Sta79a] R. STATMAN – Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science* **9** (1979), no. 1, p. 67–72.
- [Sta79b] — , The typed lambda-calculus is not elementary recursive. *Theoretical Computer Science* **9** (1979), no. 1, p. 73–81.
- [Sti02] C. STIRLING – Deciding dpda equivalence is primitive recursive. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming* (London, UK), Springer-Verlag, 2002, p. 821–832.
- [Sti06] — , A game-theoretic approach to deciding higher-order matching. In *ICALP (2)* (M. Bugliesi, B. Preneel, V. Sassone and I. Wegener, ed.), Lecture Notes in Computer Science, vol. 4052, Springer, 2006, p. 348–359.
- [Tai67] W. TAIT – Intensional interpretations of functionals of finite type I. *J. Symb. Log.* **32** (1967), no. 2, p. 198–212.
- [Zai87] M. ZAIONC – Word operation definable in the typed lambda-calculus. *Theor. Comput. Sci.* **52** (1987), p. 1–14.
- [Zai88] — , On the lambda-definable tree operations. In *Algebraic Logic and Universal Algebra in Computer Science* (C. Bergman, R. D. Maddux and D. Pigozzi, ed.), Lecture Notes in Computer Science, vol. 425, Springer, 1988, p. 279–292.
- [Zai91] — , Lambda-definability on free algebras. *Ann. Pure Appl. Logic* **51** (1991), no. 3, p. 279–300.
- [Zai95] — , Lambda representation of operations between different term algebras. *Lecture Notes in Computer Science* (1995), p. 91–105.

Index to notations

Symbolism	Meaning	Page
$FV(M)$	Set of free variables of the term M	8
$M \equiv N$	Syntactic equality of terms (modulo α -conversion)	8
$M \{N/x\}$	Capture-permitting substitution of N for x in M	8
$M [N/x]$	Substitution of N for x in M	8
\rightarrow_β	Beta-reduction	9
$s \cdot s'$	Concatenation of the (justified) sequences s and s'	27
ϵ	The empty (justified) sequence	27
$s \leq_m$	Prefix of the (justified) sequence s ending with the occurrence m	27
$\ulcorner s \urcorner$	Proponent view of a justified sequence of move	27
$\llcorner s \lrcorner$	Opponent view of a justified sequence of move	27
$\sigma; \tau$	Linear strategy composition	30
$\sigma \circ \tau$	Strategy composition	35
$\llbracket T \rrbracket$	Game denotation of a type T	36
$\llbracket M \rrbracket$	Strategy denotation of a term M	36
$C[-]$	Context with a hole denoted by $-$	38
\rightarrow_{β_s}	Safe beta-reduction	53
$[M]$	Eta-long normal form of the term M	54
$\tau(M)$	Computation tree of the term M	90
\otimes	Root of the computation tree	91

Symbolism	Meaning	Page
S^{H^+}	Subset of S consisting of the nodes hereditarily enabled by some node in H	93
$s \leq s'$	Prefix ordering for sequences	94
$\text{ip}(t)$	Immediate prefix of a justified sequence	94
$\text{jp}(t)$	Justifying prefix of a justified sequence	94
$t \upharpoonright n$	Hereditary projection of justified sequence t with respect to occurrence n	95
$t \upharpoonright^+ n$	Subterm projection of a traversal t with respect to occurrence n	104
$\text{ext}(t)$	Extension of a justified sequence of nodes	108
$\langle\langle M \rangle\rangle$	Revealed strategy denotation of a term M	117
$t \sqsubseteq t'$	Approximation ordering for trees	141
$[n_1, n_2]$	Path in a tree from node n_1 to node n_2	156
$s \sqsubseteq s'$	Subsequence relation for (justified) sequences	180
$s \geq s'$	Suffix relation for (justified) sequences	180
$\sigma \lesssim \tau$	Intrinsic preorder in the category of games \mathcal{C}	188
$\sigma \lesssim_{\mathcal{I}} \tau$	Intrinsic preorder in the category of games \mathcal{I}	188
$M \sqsubset N$	Observational preorder	194
$M \sqsubset_s N$	Observational preorder with respect to safe contexts	194

Index

Symbols	
α -convertible	8
β -reduction	9
η -long normal form	53
A	
active expressions	15
almost safe	57, 76, 155
application	57, 155
almost safe application	47
Alternation	93
alternation	27
answered	92
applicative terms	18
approximation ordering	139
arity	10
atomic types	10
B	
bad variable construct	15
beta-equality	9
beta-normal form	9
beta-redex	9
binder	90
bound	90
node	154
C	
canonical classifying category	171, 172
canonical form	14
canonical forms	16
canonical sub-ICC	169
cartesian closed category	
definition	168
generated by a typed-calculus	171
category	168
CCC	<i>see</i> cartesian closed category
Church-Rosser	10
closed term	8
co-Kleisli category	35
compact	161
typing deduction	73
compact morphisms	39
compatible	9
complete model	39
complete play	43
component	176
composite	30
composition of strategy	30
computable term	38
computation tree	88, 139
computationally adequate	38
consistent	11
consistent typing assumptions	11
constant traversal	97
contraction	9
copy-cat strategy	29
currying	34
D	
dead code	152
dead occurrences	153
dead variable elimination	153
definability	39
dummy lambda	88
E	
elementary recursive	61
enabling relation	91
eta-conversion	9
eta-long normal form	53, 54
eta-reduction	9
evaluation contexts	18
exponential	168
game	35
extension of a justified sequence of nodes	106
extensional category	186
extensional game model	41
external moves	116
F	
Finitary Idealized Algol	16
free variables	8
fresh variable	8
fully abstract	41
fully-revealed game denotation	119
G	
generalized lambda-node	145
generalized O-move in component A, B	176
generalized O-moves in component B, C	177

- generalized P-moves in component A, B . 177
 generalized P-moves in component B, C . 177
- H**
- height 32
 hereditarily enabled 91
 hereditarily justified 92
 higher-order grammar 18
 higher-order recursion scheme 19
 homogeneous 20, 169
 safe lambda calculus 46
 homogeneous incremental category 169
- I**
- ICC 168
 incremental
 justification *see* strategy P-incrementally
 justified
 morphism 168
 node binding 154
 subcategory 168
 tree binding 154
 incremental closed
 category 169
 subcategory 168
 inequationally complete 39
 inequationally fully abstract 41
 inequationally sound 39
 inhabitant 11
 initial occurrence of the thread of n 93
 initial occurrences 93
 innocence 31, *see* strategy innocent
 input-variables nodes 91
 instance
 of a type 12
 intentional category 186
 intentional game model 38
 intentionally fully-abstract 161
 interaction 30
 interaction game 115
 interaction sequence 30, 176
 interaction type 115
 interaction type tree 115
 internal language 171
 internal move 116
 intrinsic preorder 36, 186
- J**
- justified interaction sequence 116
 justified sequence of nodes 91
 justifies 92
- K**
- Kierstead terms 37
- knowing strategies 42
- L**
- large subterms 48
 left-strict 170
 legal uncovered position 117
 level 152, 173
 long O-view 106
 long safe fragment 166
 long-safe 54
- M**
- memory-cell strategy 42
 model 170
- N**
- node
 pending 92
 unanswered 92
 normal inhabitants 74
 normalizable 9
- O**
- O-incrementally justified 191
 O-view 27
 observational equivalence 38
 observational preorder 38
 one-step β -reduction 9
 Opponent 25
 order
 game 173
 move 32, 152, 173
 node 90
 type 10
 order- i finitary fragment of IA 16
 order-consistent 49
- P**
- P-incrementally justified 152
 P-view 27, 93
 interaction sequence 177
 pairing 34, 168
 path 154
 pending node 92
 pending question 31, 173
 play
 closed P-incrementally justified 175
 P-incrementally justified 175
 pointed poset 169
 pointed-poset 169
 pre-computation 88
 precongruence 171
 prime
 arena 172
 node 89

- sub-types 166
- principal deduction 12
- principal type 12
- products 168
- program 16, 192
- program contexts 192
- projection 30
- promotion 35
- Proponent 25
- pushdown automaton 21
- Q**
- quotiented category 36
- R**
- rational 170
- reachability problem 152
- reachable 153
- reduction of a traversal 100
- reflexive 9
- represented 69
- represents the pair of functions 70
- revealed strategy 115
- S**
- safe 47
 - β -reduction 52
 - IA 82
 - PCF 76
 - deduction 73
 - definition 20
 - fragment 85, 166
 - lambda calculus 46
 - lambda calculus with product 166
 - lambda calculus *à la* Church 46
 - lambda calculus *à la* Curry 46
 - pair 70
 - redex 51
 - typed-calculus 166
 - universally 47
- safe context 192
- safe program contexts 192
- safe variable typing convention 49
- semi-capture-permitting substitution 84
- semi-closed split-term 82
- set of possible moves 116
- Sierpinski game 36
- simple game 43
- simple type 10
- simply typed lambda calculus 11
- simultaneous substitution 9
- sound 39
- sound for evaluation 38
- spawn 89
- split terms-in-context 81
- star fragment 194
- store 16
- strategy 29
 - closed P-incrementally justified 163, 175
 - history-free 31
 - history-sensitive 31
 - innocent 31
 - P-incrementally justified 175
 - P-incrementally justified modulo 188
 - P-well-bracketed 173
 - well-bracketed 31
- strategy composition 30
- stratified context 59
- strongly normalizable 9
- strongly normalizing 10
- strongly safe IA 80
- sub-terms 8
- sub-traversal of the computation tree 102
- subcategory 168
- substitution
 - capture-permitting 8
 - definition 8
 - simultaneous 9
 - simultaneous capture-permitting 9
- subterm projection 102
- symmetric 9
- syntactically-revealed game denotation 119
- T**
- term 7
 - closed 8
- term-in-context 11
- terminal 168
- thread
 - in a play 185
 - in a traversal 93
 - of a move 185
- transitive 9
- traversals 94
- type 90
 - arity 10
 - binary word 69
- type substitution 12
- type-ranking function 58
- typed-calculus 166
- typing assumptions 11
- typing context 11
- typing deduction 11
- U**
- uncovered positions 117
- universality 161
- universally safe 47

universally unsafe	47
unsafe	47
unsafe type	74
untyped lambda calculus	7

V

value term	19
value-leaf	88
variable	
bound	8
free	8
fresh	8
view	
O-view	27
P-view	27, 93
view function	31
visibility	27, 93

W

weakly normalizing	10
well-behaved	98
well-bracketing	<i>see</i> strategy well-bracketed, 92
well-opened	35
word function	69