

Untyped polarized calculi

Xavier MONTILLET

July 15, 2024

0. Introduction

The goal of this thesis is to provide an introduction to polarized L calculi and to demonstrate their usefulness in studying untyped λ -calculi.

0.1. Motivation

The study of programming languages aims at making reasoning about the behavior of programs easier, and at identifying desirable properties for future programming languages. When studied formally, programming languages are equipped with a semantics, i.e. a map that assigns to each program a mathematical object that represents its behavior. The semantics then induces an equivalence relation: two programs are considered equivalent when they have the same semantics. Some aspects of the behavior of programs can be either useful or superfluous depending on the context. For example, the time a program takes to compute its result is irrelevant when reasoning about its adherence to a specification, but crucial when trying to optimize the program. This leads to some programming languages having several semantics, ranging from loose ones that account for very few aspects of the behavior and are easy to reason about, to more precise ones that account for more aspects of the behavior but are more complex.

One very desirable property of a semantics is compositionality: program fragments should also have a semantics, and the semantics of the whole program should be expressible in terms of the semantics of its fragments. For example, to get the smallest element of a list, we can write a program that sorts the list and returns the first element of the sorted list, and this works independently of the how exactly the list is sorted. The existence of a compositional semantics is a fundamental property for programming languages because it allows for large collaborative programs without requiring each individual contributor to understand every part of the program in details. The execution of a program by a computer is an inherently non-compositional process because any operation can a priori observe any part of the state of the computer. This leads to some low-level programming languages suffering from a lack of compositionality, e.g. assembly languages or those that use the `goto` statement [Dij68]. This lead to the introduction of high-level programming languages that encourage writing programs in a compositional way by disallowing the natural non-compositional ways of writing programs and providing compositional abstractions as an alternative.

One of the most popular and widely spread of those abstractions is the concept of function that allows writing program fragments that takes some inputs, and uses them to compute some output. Functions can be through of as a sort of restricted `goto` statements that eventually returns to where it started¹. This restriction makes reasoning on what happens after calling a function much easier than on what happens after a `goto` statement: we know that whatever instruction is placed after a function call will eventually be executed².

The λ -calculus [Bar84] is a bare-bones programming language used to study the expressiveness of functions. Its bare-bones nature makes studying it mathematically easier, but unsuitable to write complex programs, which is why real-world programming languages based on the λ -calculus extend it with some datatypes (e.g. numbers) and operations (e.g. addition). While those additional operations can be encoded into the λ -calculus (just like functions can be encoded with `goto` statements), the encodings can be used in more ways

¹Modulo termination.

²Again modulo termination.

[Dij68] “Letters to the Editor: Go to Statement Considered Harmful”, Dijkstra, 1968

[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

0. Introduction

than intended, which makes them harder to reason about. In the words of Robert Harper³:

The expressive power of a programming language is derived from its strictures,
not its affordances.

When trying to study programming languages with additional datatypes, a new difficulty appears: scalability. Indeed, some proofs scale quadratically in the number of datatypes, and hence become unmanageable as soon as a handful of datatypes are added. In a typed setting, it is well-known that many proofs are easier in sequent calculi than in natural deduction. In this thesis, we look at the untyped counterpart of this statement, i.e. we compare two untyped calculi: the sequent-calculus-inspired $\bar{\lambda}\mu\tilde{\mu}$ -calculus [CurHer00], and the natural-deduction-like λ -calculus. It turns out that, while the $\bar{\lambda}\mu\tilde{\mu}$ -calculus has a higher initial cost of entry, it scales much better when adding datatypes⁴, elucidates the connections between several well-known variants of the λ -calculus⁵, and suggests new better-behaved variants⁶.

³This is a quote I remember hearing at OPLSS 2019. A similar sentence can be found in an [email](#) by Robert Harper on the TYPES mailing list:

The power of a type system arises from its strictures, which can be selectively relaxed, not its affordances, which sacrifice the ability to draw sharp distinctions.

⁴Many definitions and proofs scale quadratically in the number of datatype constructors in the λ -calculus, and only linearly in the $\bar{\lambda}\mu\tilde{\mu}$ -calculus.

⁵For example, in $\bar{\lambda}\mu\tilde{\mu}$, the distinction between evaluating with the head reduction or with the weak head reductions in call-by-name can be understood as being dual to the distinction between evaluating open expressions or closed expressions in call-by-value.

⁶This includes our calculus $\lambda_p^{\rightarrow * \uparrow \otimes \oplus \parallel}$ which can be seen as a version of Call-by-push-value [Lev04; Lev06] with what Levy calls “complex values”, and our dynamically typed calculus $\lambda_N^{\mathcal{P}\mathcal{N}}$ that avoids clashes while remaining untyped.

[CurHer00] “The duality of computation”, Curien and Herbelin, 2000

0.2. Background

0.2.1. Calculi

λ -calculi and Call-by-push-value The λ -calculus [Bar84] is a well-known abstraction used to study programming languages. It has two main evaluation strategies: *call-by-name* (CBN) evaluates arguments only when they are used, while *call-by-value* (CBV) evaluates arguments immediately. Each strategy has its own advantage: call-by-name ensures that no unnecessary computations are done, while call-by-value ensures that no computations are done more than once. We write λ_N^\rightarrow and λ_V^\rightarrow for the call-by-name and call-by-value λ -calculi respectively. Each strategy induces two reductions: the strong reduction \rightarrow that can reduce anywhere in the expression, and the operational reduction \triangleright (often called the weak head reduction) that never reduces under λ -abstractions and is deterministic. While the strong reduction is the most common in the literature, the operational reduction is more closely related to real-world programming languages [Ong88; Abr90].

The call-by-name λ -calculus has been thoroughly studied [Bar84] and is well-understood. By contrast, the current understanding of the call-by-value lags behind. This is due to its study being more involved than that of call-by-name, for example requiring computation monads [Mog89; Mog91] to build models, and σ -reductions / commuting conversions to get a well-behaved reduction on open expressions [AccGue16; AccPao12; PaoRon99; GarNog16]. *Call-by-push-value* (CBPV) [Lev04; Lev06] decomposes Moggi’s computation monad as an adjunction, subsumes both call-by-name and call-by-value, and sheds some light on the interactions and differences of both strategies. CBPV also adds some datatypes (sums and pairs), and its pure fragment has been studied under the name Bang calculus [EhrGue16; BucKesRíoVis20].

The $\bar{\lambda}\mu\tilde{\mu}$ -calculus Another direction the λ -calculus has evolved in is the computational interpretation of classical logic, with continuation-passing style translations and the $\lambda\mu$ -calculus [Par92]. This eventually led to the $\bar{\lambda}\mu\tilde{\mu}$ -calculus [CurHer00], which can be understood as denoting proofs in the sequent calculus, just like λ -terms denote proofs in natural deduction. An interesting property of the $\bar{\lambda}\mu\tilde{\mu}$ -calculus is that it resembles both the

-
- [Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984
 - [Ong88] “Fully Abstract Models of the Lazy Lambda Calculus”, Ong, 1988
 - [Abr90] “The lazy lambda calculus”, Abramsky, 1990
 - [Mog89] “Computational Lambda-Calculus and Monads”, Moggi, 1989
 - [Mog91] “Notions of Computation and Monads”, Moggi, 1991
 - [AccGue16] “Open Call-by-Value”, Accattoli and Guerrieri, 2016
 - [AccPao12] “Call-by-Value Solvability, Revisited”, Accattoli and Paolini, 2012
 - [PaoRon99] “Call-by-value Solvability”, Paolini and Ronchi Della Rocca, 1999
 - [GarNog16] “No solvable lambda-value term left behind”, García-Pérez and Nogueira, 2016
 - [Lev04] *Call-By-Push-Value: A Functional/Imperative Synthesis*, Levy, 2004
 - [Lev06] “Call-by-push-value: Decomposing call-by-value and call-by-name”, Levy, 2006
 - [EhrGue16] “The Bang Calculus: An Untyped Lambda-Calculus Generalizing Call-by-Name and Call-by-Value”, Ehrhard and Guerrieri, 2016
 - [BucKesRíoVis20] “The Bang Calculus Revisited”, Bucciarelli *et al.*, 2020
 - [Par92] “ $\lambda\mu$ -Calculus: An algorithmic interpretation of classical natural deduction”, Parigot, 1992
 - [CurHer00] “The duality of computation”, Curien and Herbelin, 2000

0. Introduction

λ -calculus and the Krivine abstract machine [Kri07; CurMun10; MunSch15], which makes it suitable to study both the equational theory and the operational semantics. The full $\bar{\lambda}\mu\tilde{\mu}$ -calculus is not confluent, but two natural fragments, the call-by-name and call-by-value fragments, are. Further restricting those to their intuitionistic fragments yields calculi that correspond to the call-by-name and call-by-value λ -calculi. Since call-by-value is syntactically dual to call-by-name in the full $\bar{\lambda}\mu\tilde{\mu}$ -calculus [CurHer00; DowAri18], the additional difficulty in the study of call-by-value can be understood as stemming from the restriction to the intuitionistic fragment which breaks this symmetry.

Polarized sequent calculi Those two lines of work (CBPV and $\bar{\lambda}\mu\tilde{\mu}$) can be combined into a polarized sequent calculus LJ_p^η [CurFioMun16] or L_{int} [MunSch15]. It inherits all the advantages of CBPV (subsumes CBV and CBN without loss of confluence, allows both strategies to interact, has nice models, has nice η -rules for functions, pairs and sums, ...) and of $\bar{\lambda}\mu\tilde{\mu}$ (CBV and CBN are dual, has a simple top-level reduction that generalizes both movements of the focus inside expressions of abstract machines and commuting conversions, has classical logic built-in but can easily be restricted to intuitionistic logic, ...).

0.2.2. Solvability in arbitrary programming languages

Observational equivalence and preorder The compilation of programs often involves many optimizations where some parts of the programs are replaced by faster ones. The soundness of those transformations is studied in a compositional way by using an *observational equivalence*: two expressions, i.e. program fragments, are said to be observationally equivalent when replacing one by the other never changes the observable behavior of the encompassing program. The observational equivalence is often refined to an *observational preorder* that takes into account that some replacements are sound in one direction but not in the other, i.e. that some expressions are strictly better than others.

Operational relevance and solvability The study of the observational equivalence often relies on two notions that it preserves:

Operationally relevant expressions are those that can be used to form a program that returns a result on at least one input, i.e. those that are not completely useless. Expressions that are not operationally relevant are called *operationally irrelevant* and are often exactly the least elements of the observational preorder.

Solvable expressions are those that can be used to form programs of any chosen behavior, and expressions that are not solvable are called *unsolvable*. The intuition behind solvability is that it is an indirect way of stating that the expression computes some intermediate result

[Kri07] “A call-by-name lambda-calculus machine”, Krivine, 2007

[CurMun10] “The duality of computation under focus”, Curien and Munch-Maccagnoni, 2010

[MunSch15] “Polarised Intermediate Representation of Lambda Calculus with Sums”, Munch-Maccagnoni and Scherer, 2015

[CurHer00] “The duality of computation”, Curien and Herbelin, 2000

[DowAri18] “A tutorial on computational classical logic and the sequent calculus”, Downen and Ariola, 2018

[CurFioMun16] “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”, Curien, Fiore, and Munch-Maccagnoni, 2016

0. Introduction

that can be observed internally. Indeed, to use a solvable expression in a way that yields a chosen behavior, it suffices to observe that intermediate result, and then execute another program with the chosen behavior if the expected intermediate result was observed.

The central role of unsolvability In the call-by-name λ -calculus, the unsolvable expressions are exactly the operationally irrelevant ones. They are completely useless for writing actual programs, but are very useful for many theoretical purposes because they are a much more resilient notion of “undefined” than “being non-terminating”. Quoting from [AccPao12] (itself quoting from [Wad76]):

[...] only those expressions without normal forms which are in fact unsolvable can be regarded as being “undefined” (or better now: “totally undefined”); by contrast, all other expressions without normal forms are at least partially defined. Essentially the reason is that unsolvability is preserved by application and composition [...] which [...] is not true in general for the property of failing to have a normal form.

This leads to unsolvability being a central notion when studying λ -definability, λ -theories, the observational equivalence, or Böhm trees. When studying λ -theories (i.e. congruences on the λ -calculus that contain β -reduction), this manifests as the fact that any λ -theory that equates all expressions without a normal form is inconsistent (i.e. it is a trivial theory that identifies all expressions), while there are consistent λ -theories that equate all unsolvable expression. When studying λ -definability [dVri16] (i.e. encodings of partial recursive functions in the λ -calculus) the partiality of the function is represented by mapping inputs for which it is undefined to some “undefined” expressions of the λ -calculus. While it is possible to define “undefined” as meaning “having no normal form”, the corresponding encoding is not compositional: the encoding of the composition of two partial functions can not be encoded as the composition of the encodings. Defining “undefined” as meaning unsolvable instead allows for the definition of a compositional encoding.

Unary operational completeness In some programming languages, operational relevance and solvability are equivalent. With the intuition given above for solvability, this corresponds to saying that any (external) result of a program can be observed internally, i.e. can be used as an intermediate result. This can be thought of as being a sort of internal completeness, which we call *unary*⁷ *operational completeness*.

A programming language that does not have unary operational completeness (e.g. one where the result of a program can be an uncatchable exception) can be thought of as having either too many operationally relevant expressions or too few solvable expressions. There

⁷We call this *unary* operational completeness because it does not imply binary operational completeness, i.e. the equivalence between the corresponding binary notions: external and internal separability. See Part C.

[AccPao12] “Call-by-Value Solvability, Revisited”, Accattoli and Paolini, 2012

[Wad76] “The Relation Between Computational and Denotational Properties for Scott’s D_{infty} -Models of the Lambda-Calculus”, Wadsworth, 1976

[dVri16] “On Undefined and Meaningless in Lambda Definability”, de Vries, 2016

0. Introduction

are therefore two approaches to recovering unary operational completeness: the *restrictive*⁸ [AbrOng93] approach restrict the notion of operational relevance; and the *expansive*⁸ approach expands the notion of solvability. For example, a lack of unary operational completeness that are due to uncatchable exceptions being results can be treated either by making the uncatchable exceptions operationally irrelevant by no longer considering them as results, or by making them solvable by adding try-catch statements to the language.

Operational characterization of solvability For translations between two programming languages for which it holds, preservation of operational relevance or solvability can often be proven directly by looking at the image of reductions and normal forms through the translation, while preservation of operational irrelevance or unsolvability is often harder to prove. For example, if the translation simply embeds a programming language in its extension, operationally relevance is clearly preserved and solvability most likely is too, but this is not necessarily the case for operational irrelevance and unsolvability: the extension can add new ways of using or observing some previously operationally irrelevant or unsolvable expressions.

One way to prove that operational irrelevance or unsolvability are preserved is to use an *operational characterization of operational relevance* (resp. *solvability*), i.e. a reduction \rightsquigarrow such that weak \rightsquigarrow -normalization, strong \rightsquigarrow -normalization, and operational relevance (resp. solvability) are equivalent. Given operational characterizations \rightsquigarrow_1 and \rightsquigarrow_2 of operational relevance (resp. solvability) in the source and target programming languages, to show that a translation preserves operational irrelevance (resp. unsolvability), it suffices to show that it sends infinite \rightsquigarrow_1 reduction sequences to infinite \rightsquigarrow_2 reduction sequences, which is often fairly easy.

0.2.3. Solvability in λ -calculi

In the untyped λ -calculus, the observational equivalence is defined as only observing expressioninination, i.e. two expressions are observationally equivalent when replacing either by the other in an expressioninating (resp. diverging) program can not make the program diverge (resp. expressioninate). While this definition of observational equivalence could a priori identify too many expressions, it ends-up distinguishing any expressions we could want to use as inputs or outputs (e.g. Church encodings [Chu85] of natural numbers). A solvable expression is one that can be used to reach any expression (or equivalently any normal form), and an operationally relevant expression⁹ is one that can be used to reach at least one normal form.

⁸These two words are used in [AbrOng93] to describe ways of rectifying a “poorness of fit” between a language and its model. Here, we have no model, but we can think of the language equipped with its observational preorder as being a sort of initial model. Since the observational preorder respects external observations, the intuition of operational relevance (resp. solvability) being about external (resp. internal) results casts operational relevance (resp. solvability) as slightly more on the semantic (resp. syntactic).

⁹In the literature, the notion of operational relevance is mostly used informally, and formal notions of what we would call operational relevance are often called solvability.

[Chu85] *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*, Church, 1985

0. Introduction

Those notions of course depend on the reduction \rightsquigarrow used to evaluate the expressions, so we make this dependency explicit: given a reduction \rightsquigarrow , we write $\approx_{\rightsquigarrow}$ for the induced observational equivalence, and call \rightsquigarrow -solvability (resp. \rightsquigarrow -operational relevance) the induced notions of solvability and operational relevance. There are five main reductions that appear in the literature: \triangleright_N , $\overset{h}{\triangleright}_N$, \rightarrow_N , \triangleright_V , and \rightarrow_V . The reduction \rightarrow_N (resp. \rightarrow_V) is the strong call-by-name (resp. call-by-value) reduction, i.e. the call-by-name (resp. call-by-value) reduction that can reduce anywhere in the expression; the reduction \triangleright_N (resp. \triangleright_V) is the call-by-name (resp. call-by-value) operational reduction¹⁰ that more closely models how expressions are evaluated in a real-world call-by-name (resp. call-by-value) programming language; and the reduction $\overset{h}{\triangleright}_N$ is a call-by-name reduction such that

$$\triangleright_N \subsetneq \overset{h}{\triangleright}_N \subsetneq \rightarrow_N$$

called the (call-by-name) head reduction.

Call-by-name solvability In call-by-name, the observational equivalence $\approx_{\triangleright_N}$ induced by the call-by-name operational reduction \triangleright_N is Abramsky's one [Abr90] (in the so-called lazy λ -calculus); the observational equivalence $\approx_{\overset{h}{\triangleright}_N}$ induced by the head reduction $\overset{h}{\triangleright}_N$ is Wadsworth's one [Wad76], and the observational equivalence \approx_{\rightarrow_N} induced by the call-by-name strong reduction \rightarrow_N is Morris' one [Mor69]¹¹. It is well-known that there are strict inclusions¹² [DezGio01; Bar84; IntManPol17]

$$\approx_{\triangleright_N} \subsetneq \approx_{\rightarrow_N} \subsetneq \approx_{\overset{h}{\triangleright}_N}$$

The 6 call-by-name notions of \rightsquigarrow -solvability and \rightsquigarrow -operational relevance induced by the 3 call-by-name reductions we consider are related as depicted in Figure 0.2.1, where equivalent notions are placed in the same node, and implications between non-equivalent notions are depicted by arrows \Rightarrow . Note that both notions have an operational characterization: the stronger notion is operationally characterized by the head reduction $\overset{h}{\triangleright}_N$, while the weaker one is operationally characterized by the operational reduction \triangleright_N . Also note that using either the head reduction $\overset{h}{\triangleright}_N$ or the strong reduction \rightarrow_N yields a calculus that has unary operational completeness, but that using the operational reduction \triangleright_N does not.

The lack of unary completeness when using the operational reduction \triangleright_N is due to all λ -

¹⁰In call-by-value, to get a deterministic reduction, we need to further restrict to either left-to-right or right-to-left evaluation (depending on whether we want to evaluate functions or their arguments first). Both restrictions work for our purposes.

¹¹And can alternatively be defined by observing normal forms modulo η (or equivalently $\beta\eta$ -normal forms) [Mor69].

¹²The strictness of the inclusions can be understood as stemming from a differences of strength between their respective versions of η -conversion on Böhm trees [IntManPol17].

[Abr90] "The lazy lambda calculus", Abramsky, 1990

[Wad76] "The Relation Between Computational and Denotational Properties for Scott's D_{infty} -Models of the Lambda-Calculus", Wadsworth, 1976

[Mor69] "Lambda Calculus Models of Programming Languages", Morris, 1969

[DezGio01] "From Böhm's Theorem to Observational Equivalences: an Informal Account", Dezani-Ciancaglini and Giovannetti, 2001

[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

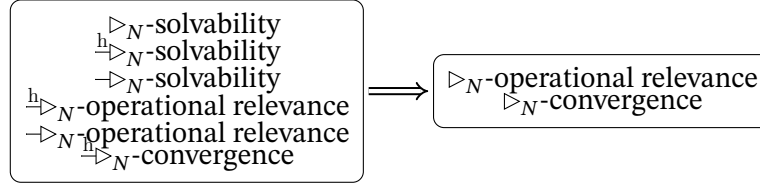


Figure 0.2.1: Notions of \rightsquigarrow -solvability and \rightsquigarrow -operational relevance in call-by-name


abstractions being \triangleright_N -operationally relevant while some of them are \triangleright_N -unsolvable¹³. The notion of order of an expression (which is more or less its arity) allows to relate both notions more precisely: \triangleright_N -operationally irrelevant expressions are exactly \triangleright_N -unsolvable expressions of order 0.

Trying to restore unary operational completeness using the restrictive approach would mean preventing (at least) some λ -abstractions from being \triangleright_N -operationally relevant, e.g. by replacing the reduction by the strong reduction \rightarrow_N or the head reduction \xrightarrow{h}_N . Using the expansive approach would mean adding a new construction that allows testing whether an expression is a λ -abstraction, e.g. an if-lambda conditional or a call-by-value let-expression. We could not find such an extension in the literature, and do not study it directly either¹⁴.

Call-by-value solvability



¹³For example, given a (closed) \triangleright_N -diverging expression T (e.g. $\Omega \triangleq \delta\delta$ where $\delta \triangleq \lambda y.xx$), the expression $\lambda x.T$ is \triangleright_N -operationally relevant (because it is \triangleright_N -normal) but \triangleright_N -unsolvable (because whenever it is given an argument, it \triangleright_N -diverges).

¹⁴However, the embeddings of the call-by-value λ -calculus into our polarized λ -calculus described in  can be understood as a way of adding such an operation.

0.3. Content



Parts A, B, and the first two chapters of part C are mostly done, but some sections still need to be cleaned up and are hidden in this draft. The last chapter of part C still has a few holes and will most likely not be part of the official thesis, but should eventually appear.

0.4. Notations

Reduction sequences A reduction \rightsquigarrow on a set \mathbf{X} is defined as being a subsets of the Cartesian square of \mathbf{X} , i.e. $\rightsquigarrow \subseteq \mathbf{X} \times \mathbf{X}$. We say that O \rightsquigarrow -reduces to O' , and write $O \rightsquigarrow O'$, when $(O, O') \in \rightsquigarrow$. We say that O is \rightsquigarrow -reducible (resp. \rightsquigarrow -normal), and write $O \rightsquigarrow$ (resp. $O \not\rightsquigarrow$) when there exists (resp. does not exist) O' such that $O \rightsquigarrow O'$, i.e. when O is (resp. is not) in the domain of \rightsquigarrow . More generally, we write $O_0 \rightsquigarrow_1 O_1 \rightsquigarrow_2 O_2 \rightsquigarrow_3 \dots \rightsquigarrow_n O_n$ for $\forall k \in \{1, \dots, n\}, O_{k-1} \rightsquigarrow O_k$, and any missing object should be understood as being quantified existentially, e.g. $O \rightsquigarrow_1 \rightsquigarrow_2 O''$ stands for $\exists O', O \rightsquigarrow_1 O' \rightsquigarrow_2 O''$. We write $\rightsquigarrow^=$ (resp. \rightsquigarrow^+ , \rightsquigarrow^*) for the reflexive (resp. transitive, reflexive transitive) closure of \rightsquigarrow . We write $O \rightsquigarrow^\circ O'$ for $O \rightsquigarrow^* O' \not\rightsquigarrow$, $O \rightsquigarrow^\circ$ for the existence of a finite maximal \rightsquigarrow -reduction sequence starting at O , and $O \rightsquigarrow^\omega$ for the existence of an infinite \rightsquigarrow -reduction sequence $O \rightsquigarrow O' \rightsquigarrow O'' \rightsquigarrow \dots$ starting at O . The inverse (as a binary relation) of a reduction \rightsquigarrow is denoted by reflecting the symbol along a vertical line: $O \rightsquigarrow O'$ is equivalent to $O' \overleftarrow{\rightsquigarrow} O$.

Main reductions We use four tip symbols for reductions: \triangleright for β -reduction, Σ for σ -reduction, \Downarrow for η -expansion, and \triangleright for an arbitrary reduction. Each symbol is combined with a vertical line to denote the operational variant of the reduction (i.e. the one relevant to study evaluation), and with a tail to denote its equational variant (i.e. the one that can reduce anywhere in the expression and is relevant to the study of the equational theory):

	β -reduction	σ -reduction	η -expansion	Arbitrary
Top-level	\triangleright	Σ	\Downarrow	\triangleright
Operational	\triangleright	Σ	\Downarrow	\triangleright
Strong	\rightarrow	\rightarrow	\rightarrow	\rightsquigarrow

Unions of some of these reductions are denoted by superimposing the symbols, e.g. the strong $\beta\sigma$ -reduction is $\rightarrow = \rightarrow \cup \rightarrow$, the strong $\beta\eta$ -reduction is $\rightarrow = \rightarrow \cup \rightarrow$, and the strong β -reduction combined with the strong η -expansion is $\rightarrow = \rightarrow \cup \rightarrow$.

Some other closures of \triangleright will be used often, and they will be denoted by \rightarrow with symbols on the tail: **t** for **top-level**, **o** for **operational**, **h** for **head**, **a** for **ahead**, **lo** for **leftmost outermost**, **s** for **strong**, and \neg for “and not”. For example, \xrightarrow{h} is the head reduction, and $\xrightarrow{s-h}$ (or $\xrightarrow{\neg h}$) is the non-head reduction.

Closure of reductions under contexts More generally, given an arbitrary set of contexts \mathbf{X} (i.e. expressions with a hole \square) and an arbitrary reduction \rightsquigarrow , we call *closure¹⁵ of \rightsquigarrow under \mathbf{X}* the reduction

$$\mathbf{X} \rightsquigarrow \stackrel{\text{def}}{=} \{(\mathbb{K} \square, \mathbb{K} \square') \mid \mathbb{K} \in \mathbf{X} \text{ and } (O, O') \in \rightsquigarrow\}$$

(where $\mathbb{K} \square$ denotes the result of plugging O in the hole \square of the context \mathbb{K}) that allows \rightsquigarrow reductions under contexts $\mathbb{K} \in \mathbf{X}$. When the reduction \rightsquigarrow is denoted by one of the four tip symbols (\triangleright , Σ , \Downarrow , or \triangleright), we also denote this closure by using the symbol for the corresponding strong reduction (i.e. \rightarrow , \rightarrow , \rightarrow , or \rightsquigarrow) and placing \mathbf{X} over its tail, e.g.

$$\xrightarrow{\mathbf{X}} = \mathbf{X} \rightarrow \quad \text{and} \quad \rightsquigarrow_{\mathbf{X}} = \mathbf{X} \rightsquigarrow$$

¹⁵For some sets \mathbf{X} , the induced operation is not really a closure because it is not idempotent. For it to be idempotent, it suffices for \mathbf{X} to be closed under composition.

0. Introduction

The previously given notations are instances of this, e.g. $\sim_{\triangleright}^{\mathbf{X}}$ is $\sim_{\triangleright}^{\mathbf{X}}$ with \mathbf{X} left implicit because it is the set of all contexts \mathbf{K} , and the symbols above the tail of \rightarrow denote sets of contexts, e.g. $\xrightarrow{\mathbf{A}}_{\triangleright}$ is the closure $\xrightarrow{\mathbf{A}}_{\triangleright}$ of \triangleright under the set \mathbf{A} of ahead contexts. Note that the negation symbol over tails denotes a set difference on contexts, e.g.

$$\xrightarrow{s \rightarrow 0}_{\triangleright} = \frac{\mathbf{K} \setminus \mathbf{O}}{\triangleright}$$

(and not a set difference on the reductions¹⁶).

Subscripts should be thought of as commuting with closures when it makes sense, e.g. \rightarrow_{let} denotes the contextual closure of $\triangleright_{\text{let}}$:

$$\rightarrow_{\text{let}} = (\mathbf{K}_N \boxed{\triangleright})_{\text{let}} = \mathbf{K}_N \boxed{\triangleright_{\text{let}}}$$

¹⁶For example, we have

$$\xrightarrow{s \rightarrow 0}_{\triangleright} \neq \xrightarrow{s}_{\triangleright} \setminus \xrightarrow{0}_{\triangleright} = \rightarrow \setminus \triangleright$$

in the λ -calculus because there can be several ways to reduce an expression to another expression:

$$(\lambda y. y)V \triangleleft (\lambda x. (\lambda y. y)x)V \xrightarrow{s \rightarrow 0}_{\triangleright} (\lambda x. x)V$$

where the \triangleleft reduction reduces the outer redex and the $\xrightarrow{s \rightarrow 0}_{\triangleright}$ one reduces the inner redex. The equation $\xrightarrow{s \rightarrow 0}_{\triangleright} = \xrightarrow{s}_{\triangleright} \setminus \xrightarrow{0}_{\triangleright}$ would hold if we thought of the reductions as being multisets that count the numbers of ways in which the reduction can happen (or used labeled transitions to allow distinguishing them), but we do not.

0.5. Table of contents

0. Introduction	2
A. Introduction to L calculi	15
I. Pure call-by-name calculi	18
II. Pure call-by-value calculi	70
B. Untyped polarized calculi	75
III. Pure polarized calculi	78
IV. Polarized calculi with pairs and sums	89
V. Polarized calculi with arbitrary constructors	103
VI. Dynamically typed polarized calculi	150
C. Solvability in polarized calculi	156
VII. Call-by-name solvability	183
VIII. Call-by-value solvability	216
IX. Polarized solvability	217
Bibliography	218

Part A.

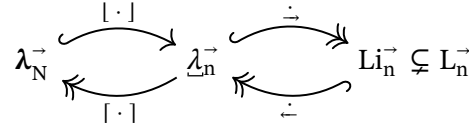
Introduction to L calculi

Part A is an introduction to the untyped $\bar{\lambda}\mu\tilde{\mu}$ -calculus [CurHer00], and more generally to calculi that look like it, which we call L-calculi. Through the Curry-Howard correspondence, the simply-typed $\bar{\lambda}\mu\tilde{\mu}$ -calculus corresponds to Gentzen’s sequent calculus for classical logic in the same way that the λ -calculus corresponds to natural deduction. Most introductions to $\bar{\lambda}\mu\tilde{\mu}$ focus on this correspondence, and sometimes mention the similarity with abstract machines. Here, we focus on the parts that are relevant to using $\bar{\lambda}\mu\tilde{\mu}$ to study the untyped λ -calculus, and in particular on the correspondence between the reductions of the call-by-name (resp. call-by-value) λ -calculus and the reductions of the call-by-name (resp. call-by-value) intuitionistic fragment of $\bar{\lambda}\mu\tilde{\mu}$.

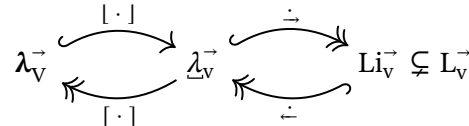
It is well-known that the operational (i.e. weak head) reduction of the call-by-name λ -calculus λ_N^\rightarrow is refined by the reduction of the Krivine abstract machine [Kri07], that makes the search for the redex explicit. The intuitionistic call-by-name fragment Li_n^\rightarrow of $\bar{\lambda}\mu\tilde{\mu}$ extends this refinement to its contextual closure, the strong reduction, that can reduce anywhere in the expression. To make understanding the call-by-name (resp. call-by-value) fragment Li_n^\rightarrow (resp. Li_v^\rightarrow) of $\bar{\lambda}\mu\tilde{\mu}$ easier, we introduce a new λ -like syntax $\underline{\lambda}_n^\rightarrow$ (resp. $\underline{\lambda}_v^\rightarrow$) for it. In this new syntax $\underline{\lambda}_n^\rightarrow$ (resp. $\underline{\lambda}_v^\rightarrow$), the reductions of the μ binder of Li_n^\rightarrow (resp. Li_v^\rightarrow) appear as a natural generalizations of the redex searching reductions of abstract machines, and of some of Regnier’s σ -reductions [Reg94].

While some advantages of using L-calculi are immediately apparent (e.g. the symmetry, and the built-in classical logic), many of their advantages only become relevant in larger calculi (e.g. those in Part B) or when studying more complex properties (e.g. those in Part C). The reader that has yet to be convinced of the usefulness of L-calculi should therefore not expect to be convinced after reading just Part A.

Content Chapter I describes the following calculi (in left-to-right order), translations¹⁷ between them, and their properties:



Chapter II describes their call-by-value counterparts:



Contribution The contribution of this part is mainly pedagogical: it provides a detailed introduction to $\bar{\lambda}\mu\tilde{\mu}$ from a new angle. Technical contributions include:

¹⁷Translations are represented by arrows with a hook \hookrightarrow when they are injective, with two heads \twoheadrightarrow when they are surjective, and with both when they are bijective.

[CurHer00] “The duality of computation”, Curien and Herbelin, 2000

[Kri07] “A call-by-name lambda-calculus machine”, Krivine, 2007

[Reg94] “Une équivalence sur les lambda-termes”, Regnier, 1994

- defining the call-by-name (resp. call-by-value) λ -calculi with focus $\underline{\lambda}_{\text{n}}^{\rightarrow}$ (resp. $\underline{\lambda}_{\text{v}}^{\rightarrow}$) as an alternative syntax for the call-by-name (resp. call-by-value) intuitionistic fragment $\text{Li}_{\text{n}}^{\rightarrow}$ (resp. $\text{Li}_{\text{v}}^{\rightarrow}$) of $\bar{\lambda}\mu\tilde{\mu}$; and
- giving a detailed description of the action of focus-inserting and focus-erasing translations $[\cdot]$ and $[\cdot]$ on reduction sequences.

I. Pure call-by-name calculi

Summary

The goal of this chapter is to recall the pure untyped call-by-name λ -calculus λ_N^\rightarrow [Bar84], the pure untyped call-by-name L calculus L_n^\rightarrow (i.e. the call-by-name fragment of $\bar{\lambda}\mu\tilde{\mu}$ [CurHer00]), and its intuitionistic fragment Li_n^\rightarrow ; to introduce the pure untyped call-by-name λ -calculus with focus $\underline{\lambda}_n^\rightarrow$ as an alternative syntax to Li_n^\rightarrow ; and to relate them via translations¹:

$$\begin{array}{ccccc} \lambda_N^\rightarrow & \xrightarrow{[\cdot]} & \underline{\lambda}_n^\rightarrow & \xrightarrow{\dot{\cdot}} & Li_n^\rightarrow \subsetneq L_n^\rightarrow \\ & \xleftarrow{[\cdot]} & & \xleftarrow{\dot{\cdot}} & \end{array}$$

In order to make the introduction of concepts more progressive, after recalling λ_N^\rightarrow , we introduce the pure untyped call-by-name λ -calculus with top-level focus $\underline{\lambda}_N^\rightarrow$ and recall the Krivine abstract machine M_N^\rightarrow [Kri07], which are simpler versions of $\underline{\lambda}_n^\rightarrow$ and Li_n^\rightarrow respectively, and are related to λ_N^\rightarrow in a similar way:

$$\begin{array}{ccccc} \lambda_N^\rightarrow & \xrightarrow{\dot{\cdot}} & \underline{\lambda}_N^\rightarrow & \xrightarrow{\dot{\cdot}} & M_N^\rightarrow \\ & \xleftarrow{[\cdot]} & & \xleftarrow{\dot{\cdot}} & \end{array}$$

In both cases, the translations $\dot{\cdot}$ and $\dot{\cdot}$ are inverses, so that up to syntax $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow (resp. $\underline{\lambda}_n^\rightarrow$ and Li_n^\rightarrow) are identical. Both the $\dot{\cdot}$ translation from λ_N^\rightarrow to $\underline{\lambda}_N^\rightarrow$ and the $[\cdot]$ translation from λ_N^\rightarrow to $\underline{\lambda}_n^\rightarrow$ add markers $\dot{\cdot}$ to make explicit where the focus is, i.e. which subexpression we are currently trying to reduce, while the $[\cdot]$ translation erases these markers. This allows to refine an operational reduction step into three simpler steps: moving the focus downwards until a redex is found, reducing the redex, and moving the focused back to the top of the expression. When looking at several successive operational reduction steps, time can be gained by not going back to the top of the expression between two steps, but instead refocusing [DanNie04], i.e. continuing the search for the next redex from where the previous redex was reduced. In $Li_n^\rightarrow / \underline{\lambda}_n^\rightarrow$, the strong reduction step can also be refined in a similar way, with focus movement replaced by a more general reduction called \rightarrow_μ , which also generalizes (some of) Regnier’s σ -reductions [Reg94].

¹Translations are represented by arrows with a hook \hookrightarrow when they are injective, with two heads \twoheadrightarrow when they are surjective, and with both when they are bijective.

[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

[CurHer00] “The duality of computation”, Curien and Herbelin, 2000

[Kri07] “A call-by-name lambda-calculus machine”, Krivine, 2007

[DanNie04] “Refocusing in Reduction Semantics”, Danvy and Nielsen, 2004

[Reg94] “Une équivelence sur les lambda-termes”, Regnier, 1994

I. Pure call-by-name calculi

Table of contents

I.1.	A pure call-by-name λ -calculus: λ_N^{\rightarrow}	20
I.2.	A pure call-by-name λ -calculus with toplevel focus: $\underline{\lambda}_N^{\rightarrow}$	27
I.3.	A pure call-by-name abstract machine: M_N^{\rightarrow}	30
I.4.	Equivalence between $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow}	35
I.5.	Translations between λ_N^{\rightarrow} and $\underline{\lambda}_N^{\rightarrow}$	46
I.6.	A pure call-by-name λ -calculus with focus: $\underline{\lambda}_n^{\rightarrow}$	51
I.7.	Translations between λ_N^{\rightarrow} and $\underline{\lambda}_n^{\rightarrow}$	61
I.8.	A pure call-by-name intuitionistic L calculus: Li_n^{\rightarrow}	62
I.9.	Equivalence between $\underline{\lambda}_n^{\rightarrow}$ and Li_n^{\rightarrow}	67
I.10.	A pure call-by-name classical L calculus: L_n^{\rightarrow}	68
I.11.	Simply-typed L calculi	69

I.1. A pure call-by-name λ -calculus: λ_N^\rightarrow

Syntax

We recall the pure untyped call-by-name λ -calculus [Bar84], which we will call λ_N^\rightarrow , in Figure I.1.1. This is the standard λ -calculus with a few minor changes to the syntax. First, we added N at all the places where polarity annotations will be needed later, e.g. to differentiate for example positive expressions T_+ from negative ones T_- or positive variables x^+ from negative ones x^- . For now, those annotations are mostly useless² (and there is no real difference between N as a subscript and N as a superscript) but we nevertheless keep them to prepare the reader for the polarized calculi. Secondly, we have let-expressions $\text{let } x^N := T_N \text{ in } U_N$, even though they behave exactly like β -redexes $(\lambda x^N. U_N) T_N$, because when translating from λ_N^\rightarrow to another calculus, the translation of $\text{let } x^N := T_N \text{ in } U_N$ is sometimes simpler than that of $(\lambda x^N. U_N) T_N$. Finally, while it is common to only refer to the objects of study as terms, we also call them values and expressions. In general, given a calculus described by a BNF grammar, we call *expressions* T (resp. *values* V , *terms* t) the elements of the syntax generated by the start non-terminal symbol (resp. same non-terminal symbols as variables x , any non-terminal symbol). In λ_N^\rightarrow , the BNF grammar only has only one non-terminal symbol, and all three names therefore denote the same objects. As is usual, application is considered to be left-associative, i.e. $T_N U_N^1 U_N^2$ stands for $(T_N U_N^1) U_N^2$. We write T_N for the set of all expressions T_N .

Figure I.1.1: Syntax of λ_N^\rightarrow

Expressions / values:
 $T_N, U_N, V_N, W_N ::= x^N \mid \text{let } x^N := T_N \text{ in } U_N$
 $\mid \lambda x^N. T_N \mid T_N U_N$

Contexts

Contexts of λ_N^\rightarrow are denoted by \mathbb{K}_N , and are generated by the BNF grammar given in Figure I.1.2.

Figure I.1.2: Contexts in λ_N^\rightarrow

Contexts:
 $\mathbb{K}_N ::= \square$
 $\mid \text{let } x^N := \mathbb{K}_N \text{ in } T_N \mid \text{let } x^N := T_N \text{ in } \mathbb{K}_N$
 $\mid \lambda x^N. \mathbb{K}_N \mid \mathbb{K}_N T_N \mid T_N \mathbb{K}_N$

²Except when looking at translations between several calculi, or skim-reading, where they serve as a reminder of which calculus we are in.

[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

I. Pure call-by-name calculi

The result of *plugging* a term T_N (resp. a context \mathbb{K}_N^0) in a context \mathbb{K}_N , i.e. the non-capture-avoiding³ substitution of \square by T_N (resp. \mathbb{K}_N^0) in \mathbb{K}_N , is denoted by $\text{plug}(\mathbb{K}_N, T_N)$ or $\mathbb{K}_N[T_N]$ (resp. $\text{plug}(\mathbb{K}_N, \mathbb{K}_N^0)$ or $\mathbb{K}_N[\mathbb{K}_N^0]$).

The weak head contexts, which we prefer calling *operational contexts* (because they allow defining the operational semantics) or *stacks* (because they correspond to stacks in abstract machines and L calculi), are defined in Figure I.1.3.

Figure I.1.3: Operational contexts in λ_N^{\rightarrow}

Operational contexts / stacks / weak head contexts:

$$\mathbf{O}_N = \mathbf{S}_N = \mathbf{\dot{S}}_N \ni \mathcal{O}_N, \mathcal{S}_N, \mathcal{\dot{S}}_N ::= \square \quad | \mathcal{O}_N T_N$$

Substitutions and disubstitutions

We write $\text{FV}(T_N)$ for the set of all free variables of T_N , and we say that a variable is *fresh* with respect to an expression when it is neither free nor bound in it. We write $T_N[V_N/x^N]$ for the usual capture avoiding substitution of x^N by V_N , denote arbitrary substitutions by σ and write $T_N[\sigma]$ for the result of applying a substitution σ to a given expression T_N .

When studying the behavior of terms (see e.g. Δ), we often want to close them via a substitution σ , and then give them arguments via a stack \mathcal{S}_N . We therefore give a name to the combination of a substitutions and a stack:

Definition I.1.1

A *disubstitution* φ is a pair $\varphi = (\sigma, \mathcal{S}_N)$ that consists of a substitution σ and a stack \mathcal{S}_N . We write $T_N[\varphi]$ for $\mathcal{S}_N[T_N[\sigma]]$.

We call these disubstitutions because they correspond to substitutions that act on both the usual variables x and on a stack variable \star in L-calculi (see Δ). We call disubstitutivity the property of being closed under disubstitutions:

Definition I.1.2

A reduction \rightsquigarrow of λ_N^{\rightarrow} is said to be:

- *substitutive* when for any substitution σ and terms T_N and T'_N , we have

$$T_N \rightsquigarrow T'_N \Rightarrow T_N[\sigma] \rightsquigarrow T'_N[\sigma]$$

³Contrary to substitutions where variable capture was avoided by renaming bound variables on the fly, e.g. $(\lambda x^N. x^N y^N)[x^N/y^N] = (\lambda z^N. z^N y^N)[x^N/y^N] = \lambda z^N. z^N x^N$, plugging does not rename anything and allows variable capture: $(\lambda x^N. x^N \square)[x^N] = \lambda x^N. x^N x^N$.

I. Pure call-by-name calculi

- *closed under stacks* when for any stack \mathbb{S}_N and terms T_N and T'_N , we have

$$T_N \rightsquigarrow T'_N \Rightarrow \mathbb{S}_N[T_N] \rightsquigarrow \mathbb{S}_N[T'_N]$$

- *disubstitutive* when for any disubstitution φ and terms T_N and T'_N , we have

$$T_N \rightsquigarrow T'_N \Rightarrow T_N[\varphi] \rightsquigarrow T'_N[\varphi]$$

Fact I.1.3

A reduction \rightsquigarrow is disubstitutive if and only if it is substitutive and closed under stacks.

Proof

\Rightarrow Take $\varphi = (\sigma, \square)$ and $\varphi = (\text{Id}, \mathbb{S}_N)$. \Leftarrow Immediate.

β -reduction

The top-level reduction \triangleright is defined in Figure I.1.4. It is the usual one (if one thinks of $\text{let } x^N := T_N \text{ in } U_N$ as being a notation for $(\lambda x^N. U_N)T_N$).

Figure I.1.4: Top-level reduction

$$\begin{aligned} \text{let } x^N := T_N \text{ in } U_N &\triangleright_{\text{let}} U_N[T_N/x^N] \\ (\lambda x^N. T_N)U_N &\triangleright_{\rightarrow} T_N[U_N/x^N] \\ \triangleright &\stackrel{\text{def}}{=} \triangleright_{\text{let}} \cup \triangleright_{\rightarrow} \end{aligned}$$

The two closures of the top-level β -reduction we are interested in for now are its operational and strong closures:

Definition I.1.4: Operational and strong reductions

The *operational reduction* \triangleright is defined as the operational closure of the top-level β -reduction \triangleright , and the *strong reduction* \rightarrow as the contextual closure of \triangleright :

$$\triangleright \stackrel{\text{def}}{=} \mathbf{O}_N \triangleright \quad \text{and} \quad \rightarrow \stackrel{\text{def}}{=} \mathbf{K}_N \triangleright$$

We write \triangleright^0 for the closure of the top-level β -reduction \triangleright under the set of non-operational contexts $\mathbf{K}_N \setminus \mathbf{O}_N$:

$$\triangleright^0 \stackrel{\text{def}}{=} (\mathbf{K}_N \setminus \mathbf{O}_N) \triangleright$$

The operational reduction \triangleright is often called the weak head reduction, but we prefer calling it the operational reduction because its main characteristic is that it induces a small-step operational semantics for the calculus, i.e. it represents evaluation. The strong reduction \rightarrow

I. Pure call-by-name calculi

should be understood as defining an equational theory $\leftrightarrow^* = (\rightarrow \cup \leftarrow)^*$ for the calculus, and it being directed helps when relating it to the operational reduction \triangleright (e.g. via the factorization $\rightarrow^* = \triangleright^* \multimap \triangleright^*$). The reductions have the properties announced in Figure ?? (see Section .2 for details).

σ -reductions

Regnier's σ -reductions [Reg94] allow commuting redexes in a way that preserves most properties of the expression:

$$\begin{aligned} (\lambda x^N. U_N) T_N V_N &\rightsquigarrow_\sigma (\lambda x^N. U_N V_N) T_N && \text{if } x^N \text{ fresh w.r.t. } V_N \\ (\lambda x^N. \lambda y^N. T_N) U_N &\rightsquigarrow_\sigma \lambda y^N. (\lambda x^N. T_N) U_N && \text{if } y^N \text{ fresh w.r.t. } U_N \end{aligned}$$

Replacing $(\lambda x^N. U_N) T_N$ by $\text{let } x^N := T_N \text{ in } U_N$ in these yields

$$\begin{aligned} (\text{let } x^N := T_N \text{ in } U_N) V_N &\rightsquigarrow_\sigma \text{let } x^N := T_N \text{ in } U_N V_N && \text{if } x^N \text{ fresh w.r.t. } V_N \\ \text{let } x^N := U_N \text{ in } \lambda y^N. T_N &\rightsquigarrow_\sigma \lambda y^N. \text{let } x^N := U_N \text{ in } T_N && \text{if } y^N \text{ fresh w.r.t. } U_N \end{aligned}$$

We only use the first of these two σ -reductions and denote it by a backwards Σ as shown in Figure I.1.5.

Figure I.1.5: Top-level σ -reduction

$$(\text{let } x^N := T_N \text{ in } U_N) V_N \Sigma \text{let } x^N := T_N \text{ in } U_N V_N \quad \text{if } x^N \text{ fresh w.r.t. } V_N$$

Definition I.1.5

We write \boxtimes for the closure of Σ under the set of simple stacks $\mathring{\mathbf{S}}_N^a$, and $\neg\boxtimes$ for the contextual closure of \boxtimes :

$$\boxtimes \stackrel{\text{def}}{=} \mathring{\mathbf{S}}_N \boxtimes \quad \text{and} \quad \neg\boxtimes \stackrel{\text{def}}{=} \mathbf{K}_N \boxtimes$$

^aWhile we have $\mathring{\mathbf{S}}_N = \mathbf{S}_N = \mathbf{O}_N$ in λ_N^{\rightarrow} , in general, we only have $\mathring{\mathbf{S}}_N \subseteq \mathbf{S}_N \subseteq \mathbf{O}_N$.

In accordance with our convention of denoting unions of reductions by superimposing their symbols, we use the notations

$$\boxtimes^{\text{ntn}} \stackrel{\text{def}}{=} \triangleright \cup \boxtimes \quad \text{and} \quad \neg\boxtimes^{\text{ntn}} \stackrel{\text{def}}{=} \rightarrow \cup \neg\boxtimes$$

In the call-by-name λ -calculus, σ -reductions are somewhat superfluous because they only relate expressions that have a common reduct:

$$(\text{let } x^N := T_N \text{ in } U_N) V_N \triangleright (U_N[T_N/x^N]) V_N \triangleleft \text{let } x^N := T_N \text{ in } U_N V_N$$

They are however very useful to make the call-by-value λ -calculus behave well on open expressions [AccGue16; AccPao12; PaoRon99], and to understand the \triangleright_μ reduction of L calculi

[Reg94] “Une équivalence sur les lambda-termes”, Regnier, 1994

[AccGue16] “Open Call-by-Value”, Accattoli and Guerrieri, 2016

[AccPao12] “Call-by-Value Solvability, Revisited”, Accattoli and Paolini, 2012

[PaoRon99] “Call-by-value Solvability”, Paolini and Ronchi Della Rocca, 1999

I. Pure call-by-name calculi

(which can be thought of as being a generalization of \boxtimes), which is why is nevertheless examine them in the call-by-name λ -calculus.

The first thing to note is that extending \triangleright by \boxtimes yields a reduction $\boxtimes = \triangleright \cup \boxtimes$ that is not deterministic:

$$\text{let } x^N := T_N \text{ in } U_N V_N \triangleright (\text{let } x^N := T_N \text{ in } U_N) V_N \triangleright_{\text{let}} (U_N[T_N/x^N]) V_N$$

This also happens in call-by-value, where we would really like to use \boxtimes to evaluate open expressions. A very common choice to avoid this problem is to simply not add \boxtimes to the operational reduction, and to only add σ -reductions $\neg\boxtimes$ in the strong reduction \rightarrow when looking at the equational theory. This of course leads to complications, e.g. requiring distinguishing σ -reduction from operational reduction in many lemmas and theorems. The reduction \triangleright_μ of Li_n^\rightarrow and Li_n^\rightarrow takes the opposite approach to recover determinism: it prevents the $\triangleright_{\text{let}}$ reduction above by disallowing the reduction of let-expressions under non-trivial operational contexts and keeps \boxtimes as part of the operational reduction!

More precisely, since \boxtimes can not be added directly to \triangleright without breaking determinism, we first restrict \triangleright and only then extend it with \boxtimes :

Definition I.1.6

The reductions \triangleright and \boxtimes are defined by

$$\triangleright \stackrel{\text{def}}{=} \triangleright_{\rightarrow} \cup \triangleright_{\text{let}} \quad \text{and} \quad \boxtimes \stackrel{\text{def}}{=} \triangleright \cup \boxtimes$$

The difference between \triangleright and \triangleright is that \triangleright allows all reductions of the shape

$$(\text{let } x^N := T_N \text{ in } U_N) V_N^1 \dots V_N^q \triangleright (U_N[T_N/x^N]) V_N^1 \dots V_N^q$$

while \triangleright only allows those of the shape

$$\text{let } x^N := T_N \text{ in } U_N \triangleright_{\text{let}} U_N[T_N/x^N]$$

i.e. those where the operational contexts $\mathcal{O}_N = \square V_N^1 \dots V_N^q$ under which the reduction happens is trivial. In particular, the $\triangleright_{\text{let}}$ reduction of the aforementioned critical pair is not allowed by \triangleright , which allows it to be deterministic:

Fact I.1.7: Determinism of \boxtimes

The reduction \boxtimes reduction is deterministic.

Proof

Both \triangleright and \boxtimes are deterministic, and they have disjoint domains.

Furthermore, the forbidden reductions

$$(\text{let } x^N := T_N \text{ in } U_N) V_N^1 \dots V_N^q \triangleright_{\text{let}} (U_N[T_N/x^N]) V_N^1 \dots V_N^q$$

I. Pure call-by-name calculi

can be simulated by

$$\begin{aligned}
 (\text{let } x^N := T_N \text{ in } U_N) V_N^1 \dots V_N^q &\sqsubseteq (\text{let } x^N := T_N \text{ in } U_N V_N^1) V_N^2 \dots V_N^q \\
 &\sqsubseteq^* \text{let } x^N := T_N \text{ in } U_N V_N^1 \dots V_N^q \\
 &\triangleright_{\text{let}} (U_N[T_N/x^N]) V_N^1 \dots V_N^q
 \end{aligned}$$

In fact, the reductions \triangleright and \sqsubseteq have the same notion of normal form, and induce the same notion of (big-step) evaluation:

Fact I.1.8: Equivalence between \triangleright^{\otimes} and \sqsubseteq^{\otimes}

- The \sqsubseteq -normal expressions are exactly the \triangleright -normal expressions:

$$T_N \sqsubseteq \Leftrightarrow T_N \triangleright$$

- The \sqsubseteq steps can be postponed at the cost of strengthening $\triangleright_{\text{let}}$ to $\triangleright_{\text{let}}^*$:

$$T_N \sqsubseteq^* T'_N \Leftrightarrow T_N \triangleright^* \sqsubseteq^* T'_N$$

- Evaluating with \sqsubseteq or \triangleright yields the same result:

$$T_N \sqsubseteq^{\otimes} T'_N \Leftrightarrow T_N \triangleright^{\otimes} T'_N$$

Proof sketch (See page 229 for details)

Immediate.

η -expansion

Another well-known and useful relation on λ -terms is η -expansion (and its symmetric, η -reduction) that relates any expressions T_N to a λ -abstraction $\lambda x^N. T_N x^N$ that has the same functional behavior, i.e. that behaves the same once given an argument. The η -expansions for λ_N^{\rightarrow} are defined in Figure I.1.6, where $\overset{\circ}{\rightarrow}$ is the standard η -expansion for functions.

Figure I.1.6: Top-level η -expansion

$$\begin{aligned}
 T_N &\overset{\circ}{\rightarrow} \lambda x^N. T_N x^N && \text{if } x^N \text{ fresh w.r.t. } T_N \\
 T_N &\overset{\circ}{\rightarrow}_{\text{let}} \text{let } x^N := T_N \text{ in } x^N
 \end{aligned}$$

We write \dashv for the contextual closure of $\overset{\circ}{\rightarrow}$, $\dashv\triangleright$ for $\dashv \cup \triangleright$, $\dashv\sqsubseteq$ for $\dashv \cup \sqsubseteq$, $\dashv\triangleright\sqsubseteq$ for $\dashv \cup \triangleright \cup \sqsubseteq$, and $\approx_{\beta\eta\sigma}$ or $\dashv\triangleright\sqsubseteq^*$ for the $\beta\eta\sigma$ -equivalence:

$$\approx_{\beta\eta\sigma} \stackrel{\text{def}}{=} \dashv\triangleright\sqsubseteq^* = (\dashv \cup \dashv\triangleright \cup \dashv\sqsubseteq \cup \dashv\triangleright\sqsubseteq)^*$$

The η -expansion for let-expressions $\overset{\circ}{\rightarrow}_{\text{let}}$ is less common, most likely because it is contained in \dashv_{let} (in call-by-name):

$$T_N \dashv_{\text{let}} \text{let } x^N := T_N \text{ in } x^N$$

I. Pure call-by-name calculi

There are other reasonable definitions of η -expansion, but all of them are contained in the $\beta\eta\sigma$ -equivalence induced by this definition of η -expansion. For example, we have

$$\mathbb{K}_N \boxed{V_N} \approx_{\beta\eta\sigma} \text{let } x^N := V_N \text{ in } \mathbb{K}_N \boxed{x^N} \quad \text{if } x^N \text{ fresh w.r.t. } \mathbb{K}_N$$

because

$$\mathbb{K}_N \boxed{V_N} <_{\text{let}} \text{let } x^N := V_N \text{ in } \mathbb{K}_N \boxed{x^N}$$

and

$$\mathbb{O}_N \boxed{T_N} \approx_{\beta\eta\sigma} \text{let } x^N := T_N \text{ in } \mathbb{O}_N \boxed{x^N} \quad \text{if } x^N \text{ fresh w.r.t. } \mathbb{O}_N$$

because

$$\mathbb{O}_N \boxed{T_N} \not\rightarrow_{\text{let}} \mathbb{O}_N \boxed{\text{let } x^N := T_N \text{ in } x^N} \not\rightarrow^* \text{let } x^N := T_N \text{ in } \mathbb{O}_N \boxed{T_N}$$

In call-by-name, all terms are values, so the first $\approx_{\beta\eta\sigma}$ -equivalence implies the second, but in call-by-value and polarized settings, neither implies the other.

I.2. A pure call-by-name λ -calculus with toplevel focus: λ_N^\rightarrow

Abstract machines use a subset of operational contexts called stacks. In general, stacks \mathcal{S}_N form a possibly strict subset of operational contexts \mathcal{O}_N , but in λ_N^\rightarrow they are exactly the same. To avoid forming intuitions that do not generalize to subsequent calculi, we call operational contexts \mathcal{O}_N stacks \mathcal{S}_N in this section. We also completely ignore let-expressions in this section because our goal is to make the comprehension of L calculi easier, and adding let-expressions at this point would not help in that regard.

Searching for the next redex

In λ_N^\rightarrow , to implement the $\triangleright_{\rightarrow}$ -reduction of a term T_N , a machine needs to decompose it as

$$T_N = \mathcal{S}_N (\lambda x^N. U_N) V_N$$

Figure I.2.1: The λ_N^\rightarrow calculus

Figure I.2.1.a: Syntax

$$\begin{array}{ll} \text{Stacks:} & \text{Configurations:} \\ \mathcal{S}_N ::= \square & \mathcal{C}_N ::= \underline{T_N} \\ | \mathcal{S}_N T_N & | \mathcal{C}_N T_N \end{array}$$

Figure I.2.1.b: Expanded descriptions

$$\begin{array}{ll} \text{Stacks (expanded):} & \text{Configurations (expanded):} \\ \mathcal{S}_N ::= \square T_N^1 \dots T_N^k & \mathcal{C}_N ::= \underline{T_N} U_N^1 \dots U_N^k \end{array}$$

Figure I.2.1.c: Operational reduction

$$\begin{array}{l} \mathcal{S}_N \boxed{T_N U_N} \xrightarrow{M} \mathcal{S}_N \boxed{\underline{T_N} U_N} \\ \mathcal{S}_N \boxed{(\lambda x^N. T_N) U_N} \xrightarrow{M} \mathcal{S}_N \boxed{T_N [U_N / x^N]} \\ \triangleright \stackrel{M}{=} \triangleright_{\rightarrow} \cup \triangleright_m^M \end{array}$$

Figure I.2.1.d: Disubstitutions

$$\begin{array}{ll} \text{Stacks:} & \text{Configurations:} \\ \mathcal{S}_N^\vee[\sigma, \mathcal{S}_N] \stackrel{\text{def}}{=} \mathcal{S}_N \boxed{\mathcal{S}_N^\vee} & \mathcal{C}_N[\sigma, \mathcal{S}_N] \stackrel{\text{def}}{=} \mathcal{S}_N \boxed{\mathcal{C}_N} \end{array}$$

I. Pure call-by-name calculi

The λ_N^{\rightarrow} calculus defined in Figure I.2.1 makes the computation of that decomposition explicit: a configuration $C_N = \mathcal{S}_N \boxed{T_N}$ represents the expression $\mathcal{S}_N \boxed{T_N}$ in which the machine is currently looking at the subexpression T_N . Initially, the machine is looking at the whole term, i.e. it starts from $\boxed{T_N}$. It then moves to the left of applications with

$$\mathcal{S}_N \boxed{T_N U_N} \xrightarrow{M} \mathcal{S}_N \boxed{T_N} U_N$$

until it reaches a λ -abstraction, at which point it reduces the β -redex with

$$\mathcal{S}_N \boxed{(\lambda x^N. T_N) U_N} \xrightarrow{M} \mathcal{S}_N \boxed{T_N [U_N / x^N]}$$

For example, the reduction

$$I_N T_N U_N \triangleright_{\rightarrow} I_N T_N U_N$$

of λ_N^{\rightarrow} becomes

$$I_N T_N U_N \xrightarrow{M} I_N T_N U_N \xrightarrow{M} I_N T_N U_N \triangleright_{\rightarrow} T_N U_N$$

in λ_N^{\rightarrow} . Note that the “move” reduction steps \xrightarrow{M} are invisible in the original calculus, while the “reduce” reduction step \xrightarrow{M} corresponds exactly to the reduction reduction step $\triangleright_{\rightarrow}$ in λ_N^{\rightarrow} .

Simulation

A top-level reduction

$$(\lambda x^N. T_N) U_N \triangleright_{\rightarrow} T_N [U_N / x^N]$$

in λ_N^{\rightarrow} becomes

$$(\lambda x^N. T_N) U_N \xrightarrow{M} (\lambda x^N. T_N) U_N \xrightarrow{M} T_N [U_N / x^N]$$

in λ_N^{\rightarrow} , and an operational reduction

$$\mathcal{S}_N \boxed{T_N} \triangleright_{\rightarrow} \mathcal{S}_N \boxed{T'_N}$$

induced by $T_N \triangleright_{\rightarrow} T'_N$ in λ_N^{\rightarrow} becomes

$$\mathcal{S}_N \boxed{T_N} \xrightarrow{M^*} \mathcal{S}_N \boxed{T_N} \xrightarrow{M} \mathcal{S}_N \boxed{T'_N} \xleftarrow{M^*} \mathcal{S}_N \boxed{T'_N}$$

in λ_N^{\rightarrow} , where the reduction sequences

$$\mathcal{S}_N \boxed{T_N} \xrightarrow{M^*} \mathcal{S}_N \boxed{T_N} \quad \text{and} \quad \mathcal{S}_N \boxed{T'_N} \xleftarrow{M^*} \mathcal{S}_N \boxed{T'_N}$$

just correspond to moving downwards through \mathcal{S}_N and do not depend on what is plugged in \mathcal{S}_N , and the $\xrightarrow{M} \xrightarrow{M}$ reduction steps correspond to the actual reduction $T_N \triangleright_{\rightarrow} T'_N$.

Refocusing

A reduction sequence

$$\mathcal{S}_N^1 \boxed{T_N V_N} \triangleright_{\rightarrow} \mathcal{S}_N^1 \boxed{T'_N} = \mathcal{S}_N^2 \boxed{U_N W_N} \triangleright_{\rightarrow} \mathcal{S}_N^2 \boxed{U'_N}$$

in λ_N^{\rightarrow} induced by

$$T_N V_N \triangleright T'_N \quad \text{and} \quad U_N W_N \triangleright U'_N$$

I. Pure call-by-name calculi

can be simulated step by step in λ_N^{\rightarrow} as

$$\begin{array}{c} \frac{\mathbb{S}_N^1 \boxed{T_N V_N}}{\nabla^z} \quad \frac{\mathbb{S}_N^1 \boxed{T'_N}}{\nabla^z} = \frac{\mathbb{S}_N^2 \boxed{U_N W_N}}{\nabla^z} \quad \frac{\mathbb{S}_N^2 \boxed{U'_N}}{\nabla^z} \\ \mathbb{S}_N^1 \boxed{T_N V_N} \xrightarrow{M} \mathbb{S}_N^1 \boxed{T'_N} \quad \mathbb{S}_N^2 \boxed{U_N W_N} \xrightarrow{M} \mathbb{S}_N^2 \boxed{U'_N} \end{array}$$

Moving the focus back to the top of the term between the two reduction steps is inefficient: instead of computing the decomposition $\mathbb{S}_N^2 \boxed{U_N W_N}$ from $\mathbb{S}_N^2 \boxed{U_N W_N}$, we could compute it from $\mathbb{S}_N^1 \boxed{T'_N}$, which is called refocusing [DanNie04]. This amounts to simplifying the reduction sequence $\xrightarrow{M^*}$ induced by one step with the reduction sequence $\xrightarrow{M^*}$ induced by the next step (using determinism of \xrightarrow{M}), which yields the shorter reduction sequence

$$\begin{array}{c} \frac{\mathbb{S}_N^1 \boxed{T_N V_N}}{\nabla^z} \quad \frac{\mathbb{S}_N^2 \boxed{U'_N}}{\nabla^z} \\ \mathbb{S}_N^1 \boxed{T_N V_N} \xrightarrow{M} \mathbb{S}_N^1 \boxed{T'_N} \xrightarrow{M^*} \mathbb{S}_N^2 \boxed{U_N W_N} \xrightarrow{M} \mathbb{S}_N^2 \boxed{U'_N} \end{array}$$

For example, for any terms T_N^1 and T_N^2 ,

$$\begin{array}{c} \frac{(\lambda x^N. I_N I_N x^N) T_N^1 T_N^2}{\nabla^z} \quad \frac{I_N I_N T_N^1 T_N^2}{\nabla^z} = \frac{I_N I_N T_N^1 T_N^2}{\nabla^z} \quad \frac{I_N T_N^1 T_N^2}{\nabla^z} = \frac{I_N T_N^1 T_N^2}{\nabla^z} \quad \frac{T_N^1 T_N^2}{\nabla^z} \\ (\lambda x^N. I_N I_N x^N) T_N^1 T_N^2 \xrightarrow{M} I_N I_N T_N^1 T_N^2 \quad I_N I_N T_N^1 T_N^2 \xrightarrow{M} I_N T_N^1 T_N^2 \quad I_N T_N^1 T_N^2 \xrightarrow{M} T_N^1 T_N^2 \end{array}$$

simplifies to

$$\begin{array}{c} \frac{(\lambda x^N. I_N I_N x^N) T_N^1 T_N^2}{\nabla^z} \quad \frac{T_N^1 T_N^2}{\nabla^z} \\ (\lambda x^N. I_N I_N x^N) T_N^1 T_N^2 \xrightarrow{M} I_N I_N T_N^1 T_N^2 \xrightarrow{M} I_N I_N T_N^1 T_N^2 \xrightarrow{M} I_N T_N^1 T_N^2 = I_N T_N^1 T_N^2 \xrightarrow{M} T_N^1 T_N^2 \end{array}$$

Properties of reductions

Disubstitutions of λ_N^{\rightarrow} are defined just like in λ_N^{\rightarrow} :

Definition I.2.1

A *disubstitution* φ is a pair $\varphi = (\sigma, \mathbb{S}_N)$ composed of a substitution σ and a stack \mathbb{S}_N . Given a configuration C_N (resp. stack \mathbb{S}_N^\vee), we write $C_N[\varphi]$ (resp. $\mathbb{S}_N^\vee[\varphi]$) or $C_N[\sigma, \mathbb{S}_N]$ (resp. $\mathbb{S}_N^\vee[\sigma, \mathbb{S}_N]$) for $\mathbb{S}_N \boxed{C_N[\sigma]}$ (resp. $\mathbb{S}_N \boxed{\mathbb{S}_N^\vee[\sigma]}$).

As announced in Figure ??, \xrightarrow{M} is deterministic, substitutive, and disubstitutive (see Section .2 for details).

[DanNie04] “Refocusing in Reduction Semantics”, Danvy and Nielsen, 2004

I.3. A pure call-by-name abstract machine: M_N^{\rightarrow}

The inside-out syntax

When implementing an abstract machine, representing $\mathcal{S}_N \boxed{T_N}$ as a tree with a marked position is suboptimal because most operations will require traversing \mathcal{S}_N , and hence takes a time linear in the depth of the hole \square in \mathcal{S}_N . It is more efficient to use a zipper [Hue97], i.e. to represent \mathcal{S}_N and T_N independently, and to represent \mathcal{S}_N in an “inside-out” fashion. More precisely, a stack \square is represented by \star , and $\mathcal{S}_N \boxed{V_N}$ by $V_N \cdot S_N$ (where S_N is the inside-out representation of \mathcal{S}_N), so that a stack

$$\mathcal{S}_N = ((\square V_N^1) \dots) V_N^k = ((\square \square V_N^k) \dots) \square V_N^1$$

is represented by

$$S_N = V_N^1 \cdot (\dots \cdot (V_N^k \cdot \star))$$

or

$$S_N = V_N^1 \cdot \dots \cdot V_N^k \cdot \star$$

with the convention that \cdot is right associative. Note that the arguments appear in the order in which they will (possibly) be needed by the computation, and that \star represents the *outside* of the context. An expression with an underlined subexpressions $\mathcal{S}_N \boxed{T_N}$ is then represented by a pair (T_N, S_N) , which we call a *configuration*⁴ and denote by $\langle T_N | S_N \rangle$, where T_N is the focused subexpression, and S_N is the inside-out representation of \mathcal{S}_N . This yields the M_N^{\rightarrow} calculus described in Figure I.3.1, a variant of the Krivine abstract machine [Kri07] that uses substitutions instead of environments and closures. Just like in the Krivine abstract machine, \xrightarrow{M}_m -reducing a term is a constant time operation thanks to the inside-out representation, but the use of substitutions in M_N^{\rightarrow} makes $\xrightarrow{M}_\rightarrow$ -reducing a term linear in the number of free occurrences of the variable, and hence less efficient than in the Krivine abstract machine.

An example reduction sequence is given in the right column of Figure I.3.2, with the corresponding reduction sequence in the left column.

As announced in Figure ??, \xrightarrow{M} is deterministic, substitutive, and disubstitutive (see Section .2 for details).

Disubstitutions

In M_N^{\rightarrow} , we also consider substitutions that act on \star^N (in addition to the usual variables x^N), which we call disubstitutions to avoid any confusion with the usual definition of substitutions:

⁴These are also sometimes called a command. In this document, we only use “configuration” for abstract machines, and keep “command” for L calculi.

[Hue97] “The Zipper”, Huet, 1997

[Kri07] “A call-by-name lambda-calculus machine”, Krivine, 2007

Figure I.3.1: The M_N^+ calculus

Figure I.3.1.a: Syntax

$$\begin{array}{ll} \text{Stacks:} & \text{Configurations:} \\ S_N ::= \star^N & C_N ::= \langle T_N | S_N \rangle \\ & | T_N \cdot S_N \end{array}$$

Figure I.3.1.b: Expanded descriptions

$$\begin{array}{ll} \text{Stacks (expanded):} & \text{Configurations (expanded):} \\ S_N ::= T_N^1 \cdot \dots \cdot T_N^q \cdot \star^N & C_N ::= \langle T_N | U_N^1 \cdot \dots \cdot U_N^q \cdot \star^N \rangle \end{array}$$

Figure I.3.1.c: Operational reduction

$$\begin{array}{l} \langle T_N U_N | S_N \rangle \xrightarrow{M} \langle T_N | U_N \cdot S_N \rangle \\ \langle \lambda x^N . T_N | U_N \cdot S_N \rangle \xrightarrow{M} \langle T_N [U_N / x^N] | S_N \rangle \\ \xrightarrow{M} \stackrel{\text{def}}{=} \xrightarrow{M} \xrightarrow{\rightarrow} \cup \xrightarrow{M}_m \end{array}$$

Figure I.3.1.d: Disubstitutions

$$\begin{array}{l} \text{Expressions:} \\ x^N[\varphi] \stackrel{\text{def}}{=} \varphi(x^N) \\ (\lambda x^N . T_N)[\varphi] \stackrel{\text{def}}{=} \lambda x^N . T_N[\varphi] \quad \text{if } x^N \text{ fresh w.r.t. } \varphi \\ (T_N U_N) \stackrel{\text{def}}{=} (T_N[\varphi])(U_N[\varphi]) \\ \text{Stacks:} \\ \star^N[\varphi] \stackrel{\text{def}}{=} \varphi(\star^N) \\ (T_N \cdot S_N)[\varphi] \stackrel{\text{def}}{=} (T_N[\varphi]) \cdot (S_N[\varphi]) \\ \text{Configurations:} \\ \langle T_N | S_N \rangle[\varphi] \stackrel{\text{def}}{=} \langle T_N[\varphi] | S_N[\varphi] \rangle \end{array}$$

<p>Figure I.3.1.e: Disubstitutions $\star^N \mapsto S_N$</p> <p>Expressions: $T_N[S_N/\star^N] = T_N$ </p> <p>Stacks: $\star^N[S_N/\star^N] = S_N$ $(T_N \cdot S_N^\vee)[S_N/\star^N] = T_N \cdot (S_N^\vee[S_N/\star^N])$ </p> <p>Configurations: $\langle T_N S_N^\vee \rangle[S_N/\star^N] = \langle T_N S_N^\vee[S_N/\star^N] \rangle$ </p>
<p>Figure I.3.1.f: Disubstitutions (simplified)</p> <p>Expressions: $T_N[\sigma, S_N/\star^N] = T_N[\sigma]$ </p> <p>Stacks: $S_N^\vee[\sigma, S_N/\star^N] = S_N^\vee[\sigma][S_N/\star^N]$ </p> <p>Configurations: $\langle T_N S_N^\vee \rangle[\sigma, S_N/\star^N] = \langle T_N[\sigma] S_N^\vee[\sigma][S_N/\star^N] \rangle$ </p>

Definition I.3.1: Disubstitutions

A *disubstitution* φ is a function of the shape $\varphi = \sigma, \star^N \mapsto S_N$, i.e. it is a substitution σ extended by $\star^N \mapsto S_N$ for some stack S_N .

One way to understand this operation is to think of \star^N as meaning “outside”, so that $C_N[S_N/\star^N]$ means replacing the “outside” of C_N by S_N . The action of disubstitutions on terms is described in Figure I.3.1d, and the special case $\varphi = \star^N \mapsto S_N$ (i.e. $\varphi = \text{Id}, \star^N \mapsto S_N$) is described in Figure I.3.1e.

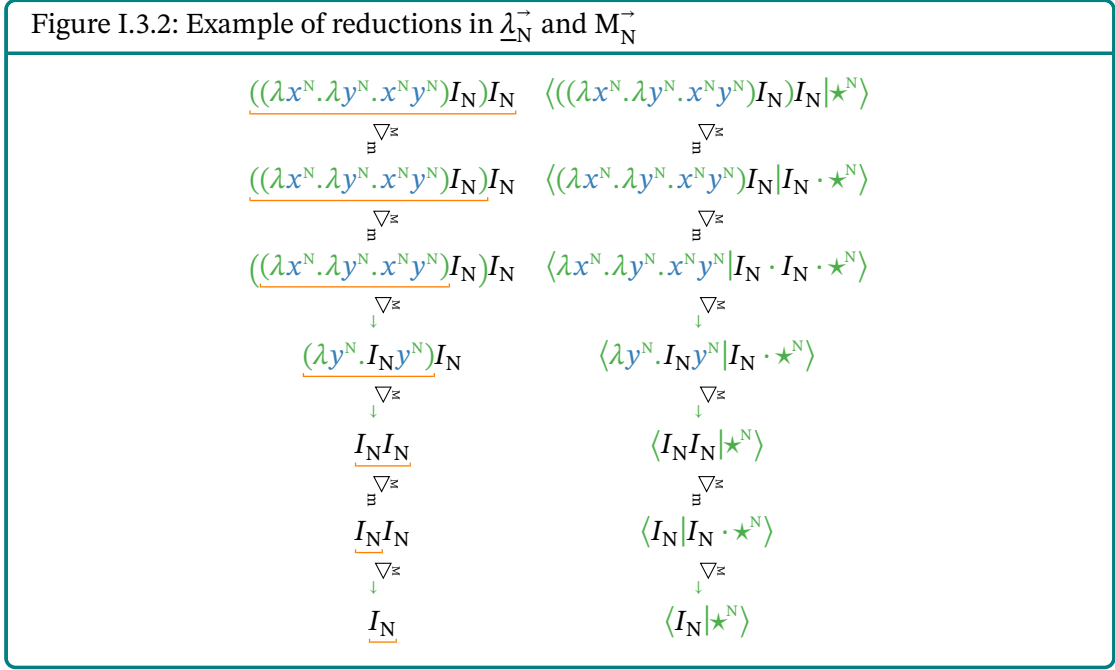
Since expressions T_N can never contain \star^N , the action of a disubstitution $\varphi = \sigma, \star^N \mapsto S_N$ can be expressed in terms of the action of the substitution σ and of the disubstitution $\star^N \mapsto S_N$:

Fact I.3.2

The equations given in Figure I.3.1f always hold.

Proof

The equation on expressions is proven by induction on T_N . The equation on stacks is prove by induction on S_N^\vee , using the equation on terms. The equation on configura-

Figure I.3.2: Example of reductions in $\underline{\lambda}_N^{\vec{}}$ and $M_N^{\vec{}}$


tions immediately follows from the equations on expressions and stacks.

Ambiguity of the ambient calculus

There is sometimes a slight ambiguity on which calculus an expression T_N lives: it could live in $\lambda_N^{\vec{}}$, $\underline{\lambda}_N^{\vec{}}$, or $M_N^{\vec{}}$. Most of the time, this ambiguity is unimportant, but it sometimes needs to be resolved:

Remark I.3.3

Translating the of disubstitutions on expressions T_N described in Figure I.3.1d to $\underline{\lambda}_N^{\vec{}}$ would yield

$$T_N[\sigma, \mathbb{S}_N] = T_N[\sigma] \quad \text{in } \underline{\lambda}_N^{\vec{}}$$

which would clash with

$$T_N[\sigma, \mathbb{S}_N] = \mathbb{S}_N[T_N[\sigma]] \quad \text{in } \lambda_N^{\vec{}}$$

This mismatch would not be that problematic because it can be trivially resolved by making the ambient calculus explicit. Furthermore, since the action of disubstitutions on expressions is uninteresting in $\underline{\lambda}_N^{\vec{}}$ and $M_N^{\vec{}}$ (because they act like substitutions), we could simply take the convention that when writing $T_N[\varphi]$, both T_N and φ live in $\lambda_N^{\vec{}}$. We nevertheless avoided redefining the action of disubstitutions on expressions in Figure I.2.1d to avoid unnecessary confusion.

I. Pure call-by-name calculi

Remark I.3.4

The \mathbf{m} in $\overset{\mathbf{m}}{\triangleright}$ is redundant (i.e. we could denote $\overset{\mathbf{m}}{\triangleright}$ by \triangleright) because \triangleright only reduces expressions, while $\overset{\mathbf{m}}{\triangleright}$ only reduces configurations, so that any reduction

$$T_N \triangleright T'_N \quad (\text{resp. } C_N \overset{\mathbf{m}}{\triangleright} C'_N)$$

necessarily happens in λ_N^\rightarrow (resp. $\underline{\lambda}_N^\rightarrow$ or M_N^\rightarrow). The remaining ambiguity between $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow is not problematic because those two calculi are basically the same (as will be shown in Section I.4).

We nevertheless keep writing $\overset{\mathbf{m}}{\triangleright}$ for the reduction of $\underline{\lambda}_N^\rightarrow$ or M_N^\rightarrow because the distinction between expressions T_N and configurations C_N may not be immediate for large terms, e.g.

$$\mathbb{S}_N^1 \left[\dots \left[\mathbb{S}_N^q \left[T_N U_N^1 \dots U_N^r \right] \right] \right] \quad \text{vs} \quad \mathbb{S}_N^1 \left[\dots \left[\mathbb{S}_N^q \left[C_N U_N^1 \dots U_N^r \right] \right] \right]$$

In $\underline{\lambda}_n^\rightarrow$ and Li_n^\rightarrow , the operational reduction will be denoted by \triangleright , and this will not lead to any semblance of ambiguity because we use lower cases letters to denote terms of $\underline{\lambda}_n^\rightarrow$ and Li_n^\rightarrow .

I.4. Equivalence between $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow}

Inside-out and outside-out descriptions

Figure I.4.1: Syntax of $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow}	
Figure I.4.1.a: Syntax of $\underline{\lambda}_N^{\rightarrow}$ (left) and outside-out description of M_N^{\rightarrow} (right)	
Stacks: $\mathbb{S}_N \ni \mathbb{S}_N ::= \square$ $\quad \mathbb{S}_N T_N$ Configurations: $\mathbb{C}_N \ni \mathbb{C}_N ::= \underline{T_N}$ $\quad \mathbb{C}_N T_N$	Stacks (outside-out): $\mathbb{S}_N \ni S_N ::= \star^N$ $\quad S_N[T_N \cdot \star^N / \star^N]$ Configurations (outside-out): $\mathbb{C}_N \ni C_N ::= \langle T_N \star^N \rangle$ $\quad C_N[T_N \cdot \star^N / \star^N]$
Figure I.4.1.b: Inside-out description of $\underline{\lambda}_N^{\rightarrow}$ (left) and syntax of M_N^{\rightarrow} (right)	
Stacks (inside-out): $\mathbb{S}_N ::= \square$ $\quad \mathbb{S}_N \boxed{T_N}$ Configurations (inside-out): $\mathbb{C}_N \ni C_N ::= \mathbb{S}_N \underline{\boxed{T_N}}$	Stacks: $S_N ::= \star^N$ $\quad T_N \cdot S_N$ Configurations: $C_N ::= \langle T_N S_N \rangle$
Figure I.4.1.c: Expanded descriptions of $\underline{\lambda}_N^{\rightarrow}$ (left) and M_N^{\rightarrow} (right)	
Stacks (expanded): $\mathbb{S}_N ::= \square T_N^1 \dots T_N^k$ Configurations (expanded): $\mathbb{C}_N ::= \underline{T_N} U_N^1 \dots U_N^k$	Stacks (expanded): $S_N ::= T_N^1$ Configurations (expanded): $C_N ::= \langle T_N U_N^1 \cdot \dots \cdot U_N^q \cdot \star^N \rangle$

The right column of Figure I.4.1b and the left column of Figure I.4.1a recall the syntaxes of M_N^{\rightarrow} and $\underline{\lambda}_N^{\rightarrow}$ respectively. Though $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow} represent the same objects, they represent them in structurally different ways. Indeed, in $\underline{\lambda}_N^{\rightarrow}$ (resp. M_N^{\rightarrow}) a stack

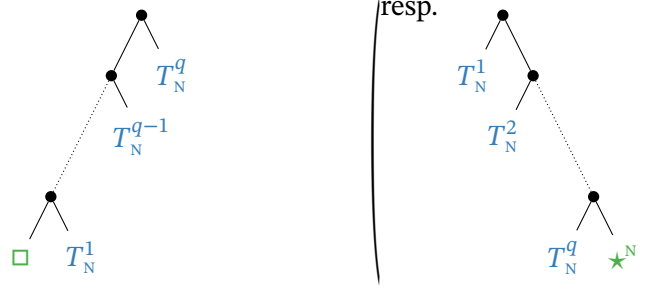
$$\mathbb{S}_N = \square T_N^1 \dots T_N^q \quad (\text{resp. } S_N = T_N^1 \cdot \dots \cdot T_N^q \cdot \star^N)$$

is implicitly parenthesized as

$$\mathbb{S}_N = ((\square T_N^1) \dots) T_N^q \quad (\text{resp. } S_N = T_N^1 \cdot (\dots \cdot (T_N^q \cdot \star^N)))$$

I. Pure call-by-name calculi

i.e. its parse tree is a left (resp. right) comb:



Taking the structure of stacks in λ_N^\rightarrow as reference, stacks of M_N^\rightarrow are therefore *inside-out*, and we call stacks of $\underline{\lambda}_N^\rightarrow$ (which are exactly stacks of λ_N^\rightarrow) *outside-out* by opposition. To make the difference in structure more apparent, we give an outside-out description of M_N^\rightarrow in the right column of Figure I.4.1a and an inside-out description of λ_N^\rightarrow in the left column of Figure I.4.1b, using an operation that substitutes \star^N by a stack S_N and plugging.

The difference between inside-out and outside-out descriptions is fairly inconsequential here because both are clearly equivalent to the expanded descriptions given in Figure I.4.1c. However, in more complex calculi, expanded descriptions become unusable. Since we only study $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow as a stepping stone towards more complex calculi, we therefore avoid using expanded descriptions.

Translations

Figure I.4.2 defines translations

$$\dot{\hookrightarrow} : \underline{\lambda}_N^\rightarrow \rightarrow M_N^\rightarrow \quad \text{and} \quad \dot{\hookleftarrow} : M_N^\rightarrow \rightarrow \underline{\lambda}_N^\rightarrow$$

It is immediate that the translation $\dot{\hookrightarrow}$ maps outside-out expressions (resp. stacks, configurations) of $\underline{\lambda}_N^\rightarrow$ to outside-out expressions (resp. stacks, configurations) of M_N^\rightarrow , and that $\dot{\hookleftarrow}$ maps inside-out expressions (resp. stacks, configurations) of M_N^\rightarrow to inside-out expressions (resp. stacks, configurations) of $\underline{\lambda}_N^\rightarrow$. To consider them as translations between $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow (i.e. between outside-out $\underline{\lambda}_N^\rightarrow$ and inside-out M_N^\rightarrow), we therefore need to show that:

- all inside-out expressions (resp. stacks, configurations) of $\underline{\lambda}_N^\rightarrow$ are outside-out expressions (resp. stacks, configurations) of $\underline{\lambda}_N^\rightarrow$; and that
- all outside-out expressions (resp. stacks, configurations) of M_N^\rightarrow are inside-out expressions (resp. stacks, configurations) of M_N^\rightarrow .

This holds thanks to the following fact:

Fact I.4.1

- In $\underline{\lambda}_N^\rightarrow$, for any stacks \mathbb{S}_N^1 and \mathbb{S}_N^2 (resp. stack \mathbb{S}_N and configuration C_N), $\mathbb{S}_N^2[\mathbb{S}_N^1]$ is a stack (resp. $\mathbb{S}_N[C_N]$ is a configuration).
- In M_N^\rightarrow , for any stacks S_N^1 and S_N^2 (resp. stack S_N and configuration C_N), $S_N^1[S_N^2/\star^N]$

Figure I.4.2: Translations $\underline{\cdot} : M_N^{\rightarrow} \rightarrow \underline{\lambda}_N^{\rightarrow}$ and $\overleftarrow{\cdot} : \underline{\lambda}_N^{\rightarrow} \rightarrow M_N^{\rightarrow}$

Figure I.4.2.a: Definition of $\underline{\cdot}$ and outside-out description of $\overleftarrow{\cdot}$

Terms:

$$\underline{T_N} \stackrel{\text{def}}{=} T_N$$

Stacks:

$$\underline{\square} \stackrel{\text{def}}{=} \star^N$$

$$\underline{\mathbb{S}_N T_N} = (\underline{\square T_N}) \underline{\mathbb{S}_N} \stackrel{\text{def}}{=} \underline{\mathbb{S}_N} [\underline{T_N} \cdot \star^N / \star^N]$$

Configurations (outside-out):

$$\underline{T_N} \stackrel{\text{def}}{=} \langle \underline{T_N} | \star^N \rangle$$

$$\underline{C_N T_N} \stackrel{\text{def}}{=} \underline{C_N} [\underline{T_N} \cdot \star^N / \star^N]$$

Terms:

$$\overleftarrow{T_N} = T_N$$

Stacks:

$$\overleftarrow{\star^N} = \square$$

$$\overleftarrow{\mathbb{S}_N [T_N \cdot \star^N / \star^N]} = (\overleftarrow{\square T_N}) \overleftarrow{\mathbb{S}_N} = \mathbb{S}_N T_N$$

Configurations:

$$\overleftarrow{\langle T_N | \star^N \rangle} = \overleftarrow{T_N}$$

$$\overleftarrow{C_N [T_N \cdot \star^N / \star^N]} = (\overleftarrow{\square T_N}) \overleftarrow{C_N} = C_N T_N$$

Figure I.4.2.b: Inside-out description of $\underline{\cdot}$ and definition of $\overleftarrow{\cdot}$

Terms:

$$\underline{T_N} = T_N$$

Stacks:

$$\underline{\square} = \star^N$$

$$\underline{\mathbb{S}_N \square T_N} = \underline{T_N} \cdot \underline{\mathbb{S}_N}$$

Configurations:

$$\underline{\mathbb{S}_N \square T_N} = \langle \underline{T_N} | \underline{\mathbb{S}_N} \rangle$$

Terms:

$$\overleftarrow{T_N} \stackrel{\text{def}}{=} T_N$$

Stacks:

$$\overleftarrow{\star^N} \stackrel{\text{def}}{=} \square$$

$$\overleftarrow{T_N \cdot \mathbb{S}_N} \stackrel{\text{def}}{=} \overleftarrow{\mathbb{S}_N} \overleftarrow{\square T_N}$$

Configurations:

$$\overleftarrow{\langle T_N | \mathbb{S}_N \rangle} \stackrel{\text{def}}{=} \overleftarrow{\mathbb{S}_N} \overleftarrow{\square T_N}$$

Figure I.4.2.c: Expanded description of $\underline{\cdot}$ and $\overleftarrow{\cdot}$

Terms:

$$\underline{T_N} = T_N$$

Stacks:

$$\underline{\square T_N^1 \dots T_N^q} = T_N^1 \cdot \dots \cdot T_N^q \cdot \star^N$$

Configurations:

$$\underline{T_N U_N^1 \dots U_N^q} = \langle \underline{T_N} | U_N^1 \cdot \dots \cdot U_N^q \cdot \star^N \rangle$$

Terms:

$$\overleftarrow{T_N} = T_N$$

Stacks:

$$\overleftarrow{T_N^1 \cdot \dots \cdot T_N^q \cdot \star^N} = \overleftarrow{\square T_N^1 \dots T_N^q}$$

Configurations:

$$\overleftarrow{\langle T_N | U_N^1 \cdot \dots \cdot U_N^q \cdot \star^N \rangle} \stackrel{\text{def}}{=} \overleftarrow{T_N} \overleftarrow{U_N^1 \dots U_N^q}$$

I. Pure call-by-name calculi

is a stack (resp. $C_N[S_N/\star^N]$ is a configuration).

Proof

- By induction on \mathbb{S}_N^2 (resp. \mathbb{S}_N).
- By induction on \mathbb{S}_N^1 (resp. C_N).

Fact I.4.2

The translation $\underline{\cdot}$ maps expressions (resp. stacks, configurations) of λ_N^\rightarrow to expressions (resp. stacks, configurations) of M_N^\rightarrow , and the translation $\underline{\cdot}$ maps (resp. stacks, configurations) of M_N^\rightarrow to expressions (resp. stacks, configurations) of λ_N^\rightarrow .

Proof

By the previous fact.

Proving that $\underline{\cdot}$ and $\underline{\cdot}$ are inverses amounts to proving that the translations distribute over plugging and substitutions of \star^N by a stack, which in turn relies on these operations inducing a monoid structure on stacks, and an action of that monoid on configurations:

Fact I.4.3

In λ_N^\rightarrow (resp. M_N^\rightarrow), the set of stacks \mathbb{S}_N has a monoid structure

$$(\mathbb{S}_N, \circ_\square, \square) \quad (\text{resp. } (\mathbb{S}_N, \circ_\star, \star^N))$$

where

$$\mathbb{S}_N^2 \circ_\square \mathbb{S}_N^1 \stackrel{\text{def}}{=} \mathbb{S}_N^2 \boxed{\mathbb{S}_N^1} \quad (\text{resp. } \mathbb{S}_N^1 \circ_\star \mathbb{S}_N^2 \stackrel{\text{def}}{=} \mathbb{S}_N^1 [S_N^2/\star^N])$$

and this monoid acts on configurations on the left (resp. on the right) via

$$\mathbb{S}_N \bullet_\square C_N \stackrel{\text{def}}{=} \mathbb{S}_N \boxed{C_N} \quad (\text{resp. } C_N \bullet_\star \mathbb{S}_N \stackrel{\text{def}}{=} C_N [S_N/\star^N])$$

In other words:

- (mon-unit) for any stack \mathbb{S}_N (resp. \mathbb{S}_N), we have

$$\square \boxed{\mathbb{S}_N} = \mathbb{S}_N = \mathbb{S}_N \square \quad (\text{resp. } \star^N [S_N/\star^N] = \mathbb{S}_N = \mathbb{S}_N [\star^N/\star^N])$$

- (mon-accoc) for any stacks \mathbb{S}_N^1 , \mathbb{S}_N^2 , and \mathbb{S}_N^3 (resp. \mathbb{S}_N^1 , \mathbb{S}_N^2 , and \mathbb{S}_N^3), we have

$$\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2 \boxed{\mathbb{S}_N^1}} = (\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2}) \boxed{\mathbb{S}_N^1} \quad (\text{resp. } \mathbb{S}_N^1 [S_N^2/\star^N] [S_N^3/\star^N] = \mathbb{S}_N^1 [S_N^2 [S_N^3/\star^N]/\star^N])$$

- (act-unit) for any configuration C_N , we have

$$\square \boxed{C_N} = C_N \quad (\text{resp. } C_N = C_N [\star^N/\star^N])$$

- (act-assoc) for any configuration C_N^1 and stacks \mathbb{S}_N^2 and \mathbb{S}_N^3 (resp. \mathbb{S}_N^2 and \mathbb{S}_N^3), we

I. Pure call-by-name calculi

have

$$\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2 \boxed{C_N^1}} = (\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2}) \boxed{C_N^1} \quad (\text{resp. } C_N^1[\mathbb{S}_N^2/\star^N][\mathbb{S}_N^3/\star^N] = C_N^1[\mathbb{S}_N^2[\mathbb{S}_N^3/\star^N]/\star^N])$$

Proof sketch (See page 230 for details)

By a few inductions.

Fact I.4.4

- For any stacks \mathbb{S}_N and \mathbb{S}_N^0 (resp. stack \mathbb{S}_N and configuration C_N) of λ_N^\rightarrow , we have

$$\boxed{\mathbb{S}_N^2 \boxed{\mathbb{S}_N^1}} = \boxed{\mathbb{S}_N^1} \boxed{\mathbb{S}_N^2/\star^N} \quad (\text{resp. } \boxed{\mathbb{S}_N} \boxed{C_N} = \boxed{C_N} \boxed{\mathbb{S}_N/\star^N})$$

- For any stacks S_N and S_N^0 (resp. stack S_N and configuration C_N) of M_N^\rightarrow , we have

$$\boxed{S_N^1} \boxed{S_N^2/\star^N} = \boxed{S_N^2} \boxed{S_N^1} \quad (\text{resp. } \boxed{C_N} \boxed{S_N/\star^N} = \boxed{S_N} \boxed{C_N})$$

Proof

By induction on \mathbb{S}_N / S_N (resp. \mathbb{S}_N^2 / S_N^2), using the previous fact.

Fact I.4.5

The translation $\underline{\cdot}$ and $\overline{\cdot}$ are each other's inverse:

- For any expression T_N (resp. stack \mathbb{S}_N , configuration C_N) of λ_N^\rightarrow , we have

$$\underline{\overline{T_N}} = T_N \quad (\text{resp. } \underline{\overline{\mathbb{S}_N}} = \mathbb{S}_N, \quad \underline{\overline{C_N}} = C_N)$$

- For any expression T_N (resp. stack S_N , configuration C_N) of M_N^\rightarrow , we have

$$\overline{\underline{T_N}} = T_N \quad (\text{resp. } \overline{\underline{S_N}} = S_N, \quad \overline{\underline{C_N}} = C_N)$$

Proof

By induction on the term, using the previous fact.

We write \rightleftharpoons for equality through these translations:

I. Pure call-by-name calculi

Definition I.4.6

We write $t_1 \Rightarrow t_2$ to state that $t_1 = t_2$, or equivalently that $t_1 = \overleftarrow{t_2}$. Whenever we write, $t_1 \Rightarrow t_2$, we implicitly assume that t_1 lives in $\underline{\lambda}_N^\rightarrow$ and that t_2 lives in M_N^\rightarrow .

With this notation, Fact I.4.4 can be reformulated as the compatibility of the operations with \Rightarrow :

Fact I.4.7

We have

$$\mathbb{S}_N^{\leftarrow 1} \Rightarrow S_N^{\rightarrow 1} \text{ and } \mathbb{S}_N^{\leftarrow 2} \Rightarrow S_N^{\rightarrow 2} \Rightarrow \mathbb{S}_N^{\leftarrow 2} \boxed{\mathbb{S}_N^{\leftarrow 1}} \Rightarrow S_N^{\rightarrow 1} [S_N^{\rightarrow 2} / \star^N]$$

and

$$\mathbb{S}_N^{\leftarrow} \Rightarrow S_N^{\rightarrow} \text{ and } \mathbb{C}_N^{\leftarrow} \Rightarrow C_N^{\rightarrow} \Rightarrow \mathbb{S}_N^{\leftarrow} \boxed{\mathbb{C}_N^{\leftarrow}} \Rightarrow C_N^{\rightarrow} [S_N^{\rightarrow} / \star^N]$$

Proof

By Fact I.4.4.

Remark I.4.8

Every notion will be defined in both $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow , and shown to be compatible with \Rightarrow (i.e. to be the same in $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow). We will often also give an equivalent outside-out (resp. inside-out) description in M_N^\rightarrow (resp. $\underline{\lambda}_N^\rightarrow$), but will leave the proof of the equivalence implicit^a. From a technical perspective, the alternative descriptions are completely superfluous, and the reader should feel free to ignore them, but we nevertheless keep them because we believe that they may have some pedagogical value.

^aThe outside-out (resp. inside-out) description in M_N^\rightarrow (resp. $\underline{\lambda}_N^\rightarrow$) will be defined as being exactly the definition in $\underline{\lambda}_N^\rightarrow$ (resp. M_N^\rightarrow) transported through \Rightarrow , so that the equivalence between the outside-out (resp. inside-out) description in M_N^\rightarrow (resp. $\underline{\lambda}_N^\rightarrow$) and the definition in M_N^\rightarrow (resp. $\underline{\lambda}_N^\rightarrow$) will immediately follow from compatibility of the definition with \Rightarrow .

For example, if we define a unary operation $f_{\leftarrow}(C_N)$ in $\underline{\lambda}_N^\rightarrow$ and the corresponding operation $f_{\rightarrow}(C_N)$ in M_N^\rightarrow , we will show that

$$C_N^1 \Rightarrow C_N^2 \Rightarrow f_{\leftarrow}(C_N^1) \Rightarrow f_{\rightarrow}(C_N^2) \quad (1)$$

The outside-out (resp. inside-out) description $f_{\rightarrow}^{\leftarrow}$ (resp. $f_{\leftarrow}^{\rightarrow}$) in M_N^\rightarrow (resp. $\underline{\lambda}_N^\rightarrow$) will be defined by starting from the equalities

$$f_{\leftarrow}^{\leftarrow}(C_N^2) = f_{\leftarrow}(\underline{C_N^2}) \quad \left(\text{resp. } f_{\rightarrow}^{\leftarrow}(C_N^1) = f_{\rightarrow}(\underline{C_N^1}) \right)$$

and possibly simplifying the right hand side. By (1), we therefore immediately get

$$f_{\leftarrow}^{\leftarrow} = f_{\rightarrow} \quad \left(\text{resp. } f_{\rightarrow}^{\leftarrow} = f_{\leftarrow} \right)$$

Substitutions

Figure I.4.3 recalls the action of substitutions on stacks and configurations of $\underline{\lambda}_N^\rightarrow$ and M_N^\rightarrow , and gives alternative descriptions. The translations are extended to substitutions pointwise:

Figure I.4.3: The action of substitutions on terms of $\lambda_N^{\vec{}} \text{ and } M_N^{\vec{}}$

Figure I.4.3.a: Definition in $\lambda_N^{\vec{}}$ (left) and outside-out description in $M_N^{\vec{}}$

<p>Stacks:</p> $\square[\sigma] \stackrel{\text{def}}{=} \square$ $(\mathbb{S}_N T_N)[\sigma] \stackrel{\text{def}}{=} (\mathbb{S}_N[\sigma])(T_N[\sigma])$ <p>Configurations:</p> $\underline{T_N}[\sigma] \stackrel{\text{def}}{=} \underline{T_N}[\sigma]$ $(C_N T_N)[\sigma] \stackrel{\text{def}}{=} (C_N[\sigma])(T_N[\sigma])$	<p>Stacks:</p> $\star^N[\sigma] = \star^N$ $(S_N[T_N \cdot \star^N / \star^N])[\sigma] = S_N[\sigma][(T_N[\sigma]) \cdot \star^N / \star^N]$ <p>Configurations:</p> $\langle T_N \star^N \rangle[\sigma] = \langle T_N[\sigma] \star^N \rangle[\sigma]$ $(C_N[T_N \cdot \star^N / \star^N])[\sigma] = C_N[\sigma][(T_N[\sigma]) \cdot \star^N / \star^N]$
---	---

Figure I.4.3.b: Inside-out description in $\lambda_N^{\vec{}}$ (left) and definition in $M_N^{\vec{}}$ (right)

<p>Stacks:</p> $\square[\sigma] = \square$ $(\mathbb{S}_N \square T_N)[\sigma] = (\mathbb{S}_N[\sigma]) \square (T_N[\sigma])$ <p>Configurations:</p> $(\mathbb{S}_N \underline{T_N})[\sigma] = (\mathbb{S}_N[\sigma]) \underline{T_N}[\sigma]$	<p>Stacks:</p> $\star^N[\sigma] \stackrel{\text{def}}{=} \star^N$ $(T_N \cdot S_N)[\sigma] \stackrel{\text{def}}{=} (T_N[\sigma]) \cdot (S_N[\sigma])$ <p>Configurations:</p> $\langle T_N S_N \rangle[\sigma] \stackrel{\text{def}}{=} \langle T_N[\sigma] S_N[\sigma] \rangle$
---	--

Figure I.4.3.c: Expanded descriptions in $\lambda_N^{\vec{}}$ (top) and $M_N^{\vec{}}$ (bottom)

<p>Stacks:</p> $(\square T_N^1 \dots T_N^q)[\sigma] = \square(T_N^1[\sigma]) \dots (T_N^q[\sigma])$ <p>Configurations:</p> $(\underline{T_N^0 T_N^1} \dots T_N^q)[\sigma] = \underline{T_N^0}[\sigma](T_N^1[\sigma]) \dots (T_N^q[\sigma])$	<p>Stacks:</p> $(T_N^1 \cdot \dots \cdot T_N^q \cdot \star^N)[\sigma] = (T_N^1[\sigma]) \cdot \dots \cdot (T_N^q[\sigma]) \cdot \star^N$ <p>Configurations:</p> $(\langle T_N^0 T_N^1 \cdot \dots \cdot T_N^q \cdot \star^N \rangle)[\sigma] = \langle T_N^0 (T_N^1[\sigma]) \cdot \dots \cdot (T_N^q[\sigma]) \cdot \star^N \rangle$
---	---

I. Pure call-by-name calculi

Definition I.4.9: Extension of \Rightarrow to substitutions

Given a substitution σ in $\underline{\lambda}_N^{\rightarrow}$ (resp. M_N^{\rightarrow}), we write $\underline{\sigma}$ (resp. $\underline{\sigma}$) for the substitution of M_N^{\rightarrow} (resp. $\underline{\lambda}_N^{\rightarrow}$) defined by

$$\underline{\sigma}(x^N) = \underline{\sigma(x^N)} \quad (\text{resp. } \underline{\sigma}(x^N) = \underline{\sigma(x^N)})$$

Given two substitutions, σ_{\leftarrow} in $\underline{\lambda}_N^{\rightarrow}$ and σ_{\rightarrow} in M_N^{\rightarrow} , we write $\sigma_{\leftarrow} \Rightarrow \sigma_{\rightarrow}$ for $\underline{\sigma}_{\leftarrow} = \sigma_{\rightarrow}$, or equivalently for $\sigma_{\leftarrow} = \underline{\sigma}_{\rightarrow}$.

Remark I.4.10

Since expressions are the same in $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow} , we have

$$\sigma_{\leftarrow} \Rightarrow \sigma_{\rightarrow} \Leftrightarrow \sigma_{\leftarrow} = \sigma_{\rightarrow}$$

We nevertheless use the notation $\sigma_{\leftarrow} \Rightarrow \sigma_{\rightarrow}$ because this will no longer be the case in $\underline{\lambda}_n^{\rightarrow}$ and $\text{Li}_n^{\rightarrow}$.

Translations distribute over the action of substitutions, i.e. substitutions are compatible with \Rightarrow :

Fact I.4.11: Compatibility of substitutions with \Rightarrow

We have

$$\sigma_{\leftarrow} \Rightarrow \sigma_{\rightarrow} \text{ and } \mathbb{S}_N^{\leftarrow} \Rightarrow S_N^{\rightarrow} \Rightarrow \mathbb{S}_N^{\leftarrow}[\sigma_{\leftarrow}] \Rightarrow S_N^{\rightarrow}[\sigma_{\rightarrow}]$$

and

$$\sigma_{\leftarrow} \Rightarrow \sigma_{\rightarrow} \text{ and } C_N^{\leftarrow} \Rightarrow C_N^{\rightarrow} \Rightarrow C_N^{\leftarrow}[\sigma_{\leftarrow}] \Rightarrow C_N^{\rightarrow}[\sigma_{\rightarrow}]$$

Proof

By induction on $\mathbb{S}_N^{\leftarrow} / C_N^{\leftarrow}$.

Disubstitutions

I. Pure call-by-name calculi

The translations are extended to disubstitutions in the expected way:

Definition I.4.12: Extension of \Rightarrow to disubstitutions

Given a disubstitution $\varphi = (\sigma, \mathbb{S}_N)$ in $\underline{\lambda}_N^{\rightarrow}$ (resp. $\varphi = \sigma, \star^N \mapsto S_N$ in M_N^{\rightarrow}), we write $\underline{\sigma}$ (resp. $\underline{\varphi}$) for the substitution of M_N^{\rightarrow} (resp. $\underline{\lambda}_N^{\rightarrow}$) defined by

$$(\underline{\sigma}, \mathbb{S}_N) = \underline{\sigma}, \star^N \mapsto \mathbb{S}_N \quad (\text{resp. } \underline{\sigma}, \star^N \mapsto S_N = (\underline{\varphi}, \mathbb{S}_N))$$

Given two disubstitutions, φ_{\leftarrow} in $\underline{\lambda}_N^{\rightarrow}$ and φ_{\rightarrow} in M_N^{\rightarrow} , we write $\varphi_{\leftarrow} \Rightarrow \varphi_{\rightarrow}$ for $\underline{\varphi}_{\leftarrow} = \varphi_{\rightarrow}$, or equivalently for $\varphi_{\leftarrow} = \underline{\varphi}_{\rightarrow}$.

The translations distribute over the translations:

Fact I.4.13: Compatibility of disubstitutions with \Rightarrow

We have

$$\varphi_{\leftarrow} \Rightarrow \varphi_{\rightarrow} \text{ and } \mathbb{S}_N^{\leftarrow} \Rightarrow S_N^{\rightarrow} \Rightarrow \mathbb{S}_N^{\leftarrow}[\varphi_{\leftarrow}] \Rightarrow S_N^{\rightarrow}[\varphi_{\rightarrow}]$$

and

$$\varphi_{\leftarrow} \Rightarrow \varphi_{\rightarrow} \text{ and } C_N^{\leftarrow} \Rightarrow C_N^{\rightarrow} \Rightarrow C_N^{\leftarrow}[\varphi_{\leftarrow}] \Rightarrow C_N^{\rightarrow}[\varphi_{\rightarrow}]$$

Proof

This holds for substitutions σ by Fact I.4.11 and for disubstitutions of the shape $\varphi_{\leftarrow} = (\text{Id}, \mathbb{S}_N)$ (resp. $\varphi_{\rightarrow} = \star^N \mapsto S_N$) by Fact I.4.7. We can therefore conclude by Fact I.3.2.

Reductions

The definitions of the operational reduction $\stackrel{M}{\triangleright}$ of $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow} are recalled in Figure I.4.4a. These two definitions correspond to each other through \Rightarrow :

Fact I.4.14: Compatibility of $\stackrel{M}{\triangleright}$ with \Rightarrow

We have

$$C_N^{\leftarrow} \Rightarrow C_N^{\rightarrow} \text{ and } C_N^{\leftarrow'} \Rightarrow C_N^{\rightarrow'} \Rightarrow (C_N^{\leftarrow} \stackrel{M}{\triangleright} C_N^{\leftarrow'} \Leftrightarrow C_N^{\rightarrow} \stackrel{M}{\triangleright} C_N^{\rightarrow'})$$

Proof

By compatibility of disubstitutions with \Rightarrow (Fact I.4.13).

Figure I.4.4: Operational reduction in λ_N^{\rightarrow} and M_N^{\rightarrow}

Figure I.4.4.a: Definition in λ_N^{\rightarrow} (left) and M_N^{\rightarrow} (right)

$$\begin{array}{l} \mathcal{S}_N \boxed{T_N U_N} \xrightarrow{M}_m \mathcal{S}_N \boxed{T_N U_N} \\ \mathcal{S}_N (\lambda x^N. T_N) U_N \xrightarrow{M}_\rightarrow \mathcal{S}_N \boxed{T_N [U_N / x^N]} \\ \xrightarrow{M} \stackrel{\text{def}}{=} \xrightarrow{M}_\rightarrow \cup \xrightarrow{M}_m \end{array}$$

$$\begin{array}{l} \langle T_N U_N | S_N \rangle \xrightarrow{M}_m \langle T_N | U_N \cdot S_N \rangle \\ \langle \lambda x^N. T_N | U_N \cdot S_N \rangle \xrightarrow{M}_\rightarrow \langle T_N [U_N / x^N] | S_N \rangle \\ \xrightarrow{M} \stackrel{\text{def}}{=} \xrightarrow{M}_\rightarrow \cup \xrightarrow{M}_m \end{array}$$

Figure I.4.4.b: Outside-out description in λ_N^{\rightarrow} (left) and M_N^{\rightarrow} (right)

$$\begin{array}{l} \overline{T_N U_N} \xrightarrow{M} \overline{T_N U_N} \\ \overline{(\lambda x^N. T_N) U_N} \xrightarrow{M} \overline{T_N [U_N / x^N]} \\ \frac{C_N \xrightarrow{M} C'_N}{C_N U_N \xrightarrow{M} C'_N U_N} \end{array}$$

$$\begin{array}{l} \overline{\langle T_N U_N | \star^N \rangle} \xrightarrow{M} \overline{\langle T_N | U_N \cdot \star^N \rangle} \\ \overline{\langle \lambda x^N. T_N | U_N \cdot \star^N \rangle} \xrightarrow{M} \overline{\langle T_N [U_N / x^N] | \star^N \rangle} \\ \frac{C_N \xrightarrow{M} C'_N}{C_N [T_N \cdot \star^N / \star^N] \xrightarrow{M} C'_N [T_N \cdot \star^N / \star^N]} \end{array}$$

Figure I.4.4.c: Inside-out description in λ_N^{\rightarrow} (left) and M_N^{\rightarrow} (right)

$$\begin{array}{l} \overline{\mathcal{S}_N \boxed{T_N U_N}} \xrightarrow{M} \overline{\mathcal{S}_N \boxed{T_N U_N}} \\ \overline{\mathcal{S}_N (\lambda x^N. T_N) U_N} \xrightarrow{M} \overline{\mathcal{S}_N \boxed{T_N [U_N / x^N]}} \end{array}$$

$$\begin{array}{l} \overline{\langle T_N U_N | S_N \rangle} \xrightarrow{M} \overline{\langle T_N | U_N \cdot S_N \rangle} \\ \overline{\langle \lambda x^N. T_N | U_N \cdot S_N \rangle} \xrightarrow{M} \overline{\langle T_N [U_N / x^N] | S_N \rangle} \end{array}$$

I. Pure call-by-name calculi

Figures I.4.4b and I.4.4c give inside-out and outside-out descriptions via inference rules, which are of course equivalent:

Fact I.4.15

In λ_N^{\rightarrow} (resp. M_N^{\rightarrow}), the definition of \triangleright^M (Figure I.4.4a) is equivalent to its outside-out description (Figure I.4.4b), and to its inside-out description (Figure I.4.4c).

Proof

- definition \Leftrightarrow inside-out Trivial.
- definition \Leftrightarrow outside-out Thinking of \mathcal{S}_N (resp. S_N) as being an outside-out stack, the \Rightarrow implication holds by induction on the stack, and the \Leftarrow holds by induction on the derivation.

The fact that stacks should be inside-out in Figure I.4.4c would be clearer if we were to give a similar inside-out description of the strong reduction \rightarrow . For example, we would have

$$\frac{U_N \rightarrow U'_N}{\frac{\mathcal{S}_N \boxed{U_N} \rightarrow \mathcal{S}_N \boxed{U'_N}}{\mathcal{S}_N \boxed{T_N U_N} \rightarrow \mathcal{S}_N \boxed{T_N U'_N}}}$$

An inside-out description of \rightarrow in λ_n^{\rightarrow} and Li_n^{\rightarrow} is given in [A.1](#). We do not give one in λ_N^{\rightarrow} and M_N^{\rightarrow} because these calculi are not the right setting to study the strong reduction \rightarrow .

I.5. Translations between λ_N^{\rightarrow} and $\underline{\lambda}_N^{\rightarrow}$

Focus insertion and erasure

We start by defining the focus-erasing translation $\llbracket \cdot \rrbracket$ from $\underline{\lambda}_N^{\rightarrow}$ to λ_N^{\rightarrow} in Figure I.5.1 that removes the underlinement in \underline{C}_N , and the corresponding translation from M_N^{\rightarrow} to λ_N^{\rightarrow} , which we denote by the same symbol.

Figure I.5.1: The focus-erasing translations $\llbracket \cdot \rrbracket : \underline{\lambda}_N^{\rightarrow} \rightarrow \lambda_N^{\rightarrow}$ and $\llbracket \cdot \rrbracket : M_N^{\rightarrow} \rightarrow \lambda_N^{\rightarrow}$	
Figure I.5.1.a: Definition in $\underline{\lambda}_N^{\rightarrow}$ (left) and outside-out description in M_N^{\rightarrow} (right)	
$\llbracket \underline{T}_N \rrbracket \stackrel{\text{def}}{=} T_N$ $\llbracket \underline{C}_N \underline{T}_N \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{C}_N \rrbracket T_N$	$\llbracket \langle \underline{T}_N \star^N \rangle \rrbracket \stackrel{\text{def}}{=} T_N$ $\llbracket \underline{C}_N [\underline{T}_N \cdot \star^N / \star^N] \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{C}_N \rrbracket [\underline{T}_N \cdot \star^N / \star^N]$
Figure I.5.1.b: Inside-out description in $\underline{\lambda}_N^{\rightarrow}$ (left) and definition in M_N^{\rightarrow} (right)	
$\llbracket \underline{S}_N \underline{T}_N \rrbracket \stackrel{\text{def}}{=} \underline{S}_N \llbracket \underline{T}_N \rrbracket$	$\llbracket \langle \underline{T}_N \underline{S}_N \rangle \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{S}_N \rrbracket \llbracket \underline{T}_N \rrbracket$

These two translations of course correspond to each other through \Rightarrow :

Fact I.5.1
<p>We have</p> $C_N^{\leftarrow} \Rightarrow C_N^{\rightarrow} \Rightarrow \llbracket C_N^{\leftarrow} \rrbracket = \llbracket C_N^{\rightarrow} \rrbracket$

Proof

Immediate.

Erasing the underlinement preserves disubstitutions:

Fact I.5.2
<p>For any configuration \underline{C}_N and disubstitution φ, $\llbracket \underline{C}_N[\varphi] \rrbracket = \llbracket \underline{C}_N \rrbracket[\varphi]$.</p>

Proof

Immediate.

The focus-erasing translation is a left inverse of $\underline{\cdot}$ and $\langle \cdot | \star^N \rangle$:

I. Pure call-by-name calculi

Fact I.5.3

For any expression T_N , we have

$$[\underline{T_N}] = T_N \quad \text{and} \quad [\langle T_N | \star^N \rangle] = T_N$$

Proof

Immediate.

Composing these two maps in the opposite order yields the identity only up to $\stackrel{M}{\triangleright}_m$ reductions:

Fact I.5.4

For any configuration C_N of $\underline{\lambda}_N^\rightarrow$ (resp. M_N^\rightarrow), we have

$$[\underline{C_N}] \stackrel{M}{\triangleright}_m^* C_N \quad (\text{resp. } \langle [\underline{C_N}] | \star^N \rangle \stackrel{M}{\triangleright}_m^* C_N)$$

i.e. for any stack S_N (resp. S_N) and term T_N of $\underline{\lambda}_N^\rightarrow$ (resp. M_N^\rightarrow), we have

$$\underline{S_N} [\underline{T_N}] \stackrel{M}{\triangleright}_m^* S_N [\underline{T_N}] \quad (\text{resp. } \langle \underline{S_N} [\underline{T_N}] | \star^N \rangle \stackrel{M}{\triangleright}_m^* \langle S_N | S_N \rangle)$$

Proof

We have

$$[\underline{S_N} [\underline{T_N}]] = \underline{S_N} [\underline{T_N}] \stackrel{M}{\triangleright}_m^* S_N [\underline{T_N}] \quad (\text{resp. } \langle [\underline{S_N} [\underline{T_N}]] | \star^N \rangle = \langle \underline{S_N} [\underline{T_N}] | \star^N \rangle \stackrel{M}{\triangleright}_m^* \langle S_N | S_N \rangle)$$

where the equality is given by Fact I.5.2, and the $\stackrel{M}{\triangleright}_m^*$ reduction sequence is obtained by induction on S_N (resp. S_N).

Reductions through focus erasure

Since $\stackrel{M}{\triangleright}_m$ reductions only move focus, they are erased by $[\cdot]$:

Fact I.5.5

If $C_N \stackrel{M}{\triangleright}_m C'_N$ then $[C_N] = [C'_N]$.

Proof

By Fact I.5.2.

Conversely, two configurations whose image by $[\cdot]$ are equal are related by \triangleright_m steps:

I. Pure call-by-name calculi

Fact I.5.6

If $[C_N] = [C'_N]$ then either $C_N \xrightarrow{M}_m^* C'_N$ or $C_N \xrightarrow{M}_m^* C'_N$.

Proof

Applying Fact I.5.4 to both C_N and C'_N yields

$$C_N \xrightarrow{M}_m^j [C_N] = [C'_N] \xrightarrow{M}_m^k C'_N$$

By determinism of \xrightarrow{M} , we can simplify this to get either $j = 0$ or $k = 0$.

The $\xrightarrow{M}_\rightarrow$ reductions of M_N^\rightarrow are preserved by focus erasure:

Fact I.5.7

If $C_N \xrightarrow{M}_\rightarrow C'_N$ then $[C_N] \xrightarrow{M}_\rightarrow [C'_N]$.

Proof

By Fact I.5.2.

Reductions through focus insertion

A top-level reduction $\xrightarrow{M}_\rightarrow$ in λ_N^\rightarrow becomes $\xrightarrow{M}_m^M \xrightarrow{M}_\rightarrow$ in λ_N^\rightarrow :

Fact I.5.8

If $T_N \xrightarrow{M}_\rightarrow T'_N$ then $\underline{T_N} \xrightarrow{M}_m^M \xrightarrow{M}_\rightarrow \underline{T'_N}$.

Proof

We have

$$(\underline{\lambda x^N. T_N}) \underline{U_N} \xrightarrow{M}_m (\underline{\lambda x^N. T_N}) \underline{U_N} \xrightarrow{M}_\rightarrow \underline{T_N[U_N/x^N]}$$

The search for the redex is represented by a sequence of \xrightarrow{M}_m reductions. The \xrightarrow{M}_m reduction can find redexes under all operational contexts:

Fact I.5.9

For any operational contexts \mathcal{O}_N and expression T_N , $\mathcal{O}_N \underline{T_N} \xrightarrow{M}_m^* \mathcal{O}_N \underline{T_N}$.

I. Pure call-by-name calculi

Proof

Since all operational contexts \mathcal{O}_N are stacks \mathcal{S}_N , this is just Fact I.5.4.

We can therefore simulate $\triangleright_{\rightarrow}$ steps as follows:

Fact I.5.10

If $T_N \triangleright_{\rightarrow} T'_N$ then $\underline{T_N} \xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m} \xrightarrow{\triangleleft_m^*} \underline{T'_N}$.

Proof

We have

$$\mathcal{O}_N[\underline{(\lambda x^N. T_N) U_N}] \xrightarrow{\triangleright_m^*} \mathcal{O}_N[\underline{(\lambda x^N. T_N) U_N}] \xrightarrow{\triangleright_m} \mathcal{O}_N[\underline{T_N[U_N/x^N]}] \xrightarrow{\triangleleft_m^*} \mathcal{O}_N[\underline{T_N[U_N/x^N]}]$$

where the $\xrightarrow{\triangleright_m^*}$ and $\xrightarrow{\triangleleft_m^*}$ reduction sequences are by Fact I.5.9.

We then look at sequences of reductions $\triangleright_{\rightarrow}^*$. The fact that the abstract machine does not need to go back to the top of the expression, sometimes called refocusing [DanNie04], can be expressed as follows:

Fact I.5.11

If $T_N \triangleright_{\rightarrow}^k T'_N$ then $\underline{T_N} (\xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m})^k \xrightarrow{\triangleleft_m^*} \underline{T'_N}$.

Proof

The previous fact gives us

$$\underline{T_N} (\xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m})^k \xrightarrow{\triangleleft_m^*} \underline{T'_N}$$

which can be rewritten (for $k \geq 1$) as

$$\underline{T_N} \xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m} (\xrightarrow{\triangleleft_m^*} \xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m})^{k-1} \xrightarrow{\triangleleft_m^*} \underline{T'_N}$$

Since $\xrightarrow{\triangleright_m}$ is deterministic, this implies

$$\underline{T_N} \xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m} ((\xrightarrow{\triangleleft_m^*} \cup \xrightarrow{\triangleright_m^*}) \xrightarrow{\triangleright_m})^{k-1} \xrightarrow{\triangleleft_m^*} \underline{T'_N}$$

and since $\xrightarrow{\triangleright_m}$ and $\xrightarrow{\triangleright_m}$ have disjoint domains, we get

$$\underline{T_N} \xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m} (\xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m})^{k-1} \xrightarrow{\triangleleft_m^*} \underline{T'_N}$$

i.e.

$$\underline{T_N} (\xrightarrow{\triangleright_m^*} \xrightarrow{\triangleright_m})^k \xrightarrow{\triangleleft_m^*} \underline{T'_N}$$

Since $\xrightarrow{\triangleright_m}$ steps are erased by $\llbracket \cdot \rrbracket$, the above implication is an equivalence, so that:

[DanNie04] “Refocusing in Reduction Semantics”, Danvy and Nielsen, 2004

I. Pure call-by-name calculi

Proposition I.5.12

For any configurations C_N and C'_N , we have

$$[C_N] \triangleright^{\oplus} \Leftrightarrow C_N \triangleright^{\oplus}$$

In particular, for any expressions T_N and T'_N , we have

$$T_N \triangleright^{\oplus} \Leftrightarrow \underline{T_N} \triangleright^{\oplus}$$

Proof

- \Rightarrow Suppose that $T_N \triangleright^{\oplus} T'_N$. By the previous fact, we have

$$\underline{T_N} (\triangleright_m^* \triangleright_{\rightarrow}^*)^k C'_N \triangleleft_m^* \underline{T'_N}$$

for some C'_N . Since \triangleright_m^* is strongly normalizing (because the depth of the expression minus the depth of $\underline{\cdot}$ in it strictly decreases at each \triangleright_m^* step), we can find C''_N such that

$$\underline{T_N} (\triangleright_m^* \triangleright_{\rightarrow}^*)^k C'_N \triangleright_m^{\oplus} C''_N \triangleleft_m^{\oplus} \underline{T'_N}$$

It now suffices to show that $C''_N \triangleright_{\rightarrow}^*$. By Fact I.5.5, we have

$$[C''_N] = [\underline{T'_N}] = T'_N$$

so that having $C''_N \triangleright_{\rightarrow}^*$ would contradict the hypothesis $T'_N \not\triangleright_{\rightarrow}^*$ by Fact I.5.7.

- \Leftarrow Suppose that $C_N \triangleright^{\oplus} C'_N$. By Facts I.5.5 and I.5.7, we have $[C_N] \triangleright^* [C'_N]$. Since $C'_N \not\triangleright$, Fact I.5.10 allows to conclude that $[C'_N] \triangleright_{\rightarrow}^*$.

I.6. A pure call-by-name λ -calculus with focus: λ_n^\rightarrow

In this section, we introduce the pure call-by-name λ -calculus with focus λ_n^\rightarrow , which is the λ -like syntax for the calculus we are really interested in: Li_n^\rightarrow . While it is suboptimal from a technical standpoint, we expect the λ_n^\rightarrow syntax to make understanding how Li_n^\rightarrow computes easier.

Section I.6.1 refines M_N^\rightarrow to allow decomposing the strong reduction, Section I.6.2 add lets expressions, and Section I.6.3 describes the actual λ_n^\rightarrow calculus.

I.6.1. The simple fragment of the naive λ_n^\rightarrow calculus

Decomposing the strong reduction

As we have seen in Section I.5, the operational reduction of λ_n^\rightarrow and M_N^\rightarrow refines that of λ_N^\rightarrow by making the implicit \triangleright_m steps explicit. This has the unfortunate consequence of damaging the relationship between the operational reduction \triangleright and the strong reduction \rightarrow . For example, the expression $(\lambda x^N. I_N W_N) V_N$ (where $I_N = \lambda y^N. y^N$) of λ_N^\rightarrow is represented by $(\lambda x^N. I_N W_N) V_N$

Figure I.6.1: Example of strong reduction in subterms of abstract machines

Figure I.6.1.a: Example in λ_N^\rightarrow

$$\begin{array}{c}
 (\lambda x^N. I_N W_N) V_N \triangleright_m (\lambda x^N. I_N W_N) V_N \triangleright \rightarrow I_N W_N[V_N/x^N] \\
 \downarrow \quad \quad \quad \downarrow \quad \quad \quad \downarrow \\
 (\lambda x^N. W_N) V_N \triangleright_m (\lambda x^N. W_N) V_N \triangleright \rightarrow W_N[V_N/x^N]
 \end{array}$$

Figure I.6.1.b: Example in λ_n^\rightarrow

$$\begin{array}{c}
 (\lambda x^n. I_n w_n) v_n \triangleright_m (\lambda x^n. I_n w_n) v_n \triangleright \rightarrow I_n w_n[v_n/x^n] \\
 \downarrow \quad \quad \quad \downarrow \quad \quad \quad \downarrow \\
 (\lambda x^n. I_n w_n) v_n \triangleright_m (\lambda x^n. I_n w_n) v_n \triangleright \rightarrow I_n w_n[v_n/x^n] \\
 \downarrow \quad \quad \quad \downarrow \quad \quad \quad \downarrow \\
 (\lambda x^n. w_n) v_n \triangleright_m (\lambda x^n. w_n) v_n \triangleright \rightarrow w_n[v_n/x^n]
 \end{array}$$

I. Pure call-by-name calculi

in λ_N^\rightarrow and reduces as shown in Figure I.6.1a, where the inner reduction $I_N W_N \triangleright W_N$ can not be refined as two steps $\triangleright_{\rightarrow} \triangleright_m$ because $I_N W_N$ has no underlined subterm.

A naive attempt at modifying λ_N^\rightarrow to allow to explicitly moving the focus in subterms can be found in Figure I.6.2. Just like in λ_N^\rightarrow , commands c_n (which correspond to configurations of λ_N^\rightarrow) are computations that can be reduced / evaluated by the operational reduction \triangleright , while an expression t_n is only part of a computation meant to be combined with a stack \mathbb{S}_n (or an evaluation context \mathcal{E}_n in the full calculus) to form a command. Of course, any expression t_n can be turned into a command by making it interact with the trivial stack \square , which yields \underline{t}_n . The distinction between commands and simple commands will become relevant later when we add let-expressions (and the distinction between stacks and simple stacks will become

Figure I.6.2: The simple fragment of the **naive** call-by-name λ -calculus with focus λ_N^\rightarrow

Figure I.6.2.a: Syntax (**naive**)

Expressions / values:	Stacks / simple stacks:
$t_n, u_n, v_n, w_n ::= x^n \mid c_n$	$\mathbb{S}_n, \mathbb{S}'_n ::= \square$
$\mid \lambda x^n. c_n$	$\mid \mathbb{S}_n t_n$
Commands:	Simple commands:
$c_n ::= \mathring{c}_n$	$\mathring{c}_n ::= \underline{t}_n$
	$\mid \mathring{c}_n t_n$

Figure I.6.2.b: Expanded description of commands and stacks (**naive**)

Simple commands:	Stacks / simple stacks:
$\mathring{c}_n ::= \underline{t}_n u_n^1 \dots u_n^q$	$\mathbb{S}_n, \mathbb{S}'_n ::= \square u_n^1 \dots u_n^q$

Figure I.6.2.c: Operational reduction (**naive**)

$$\begin{aligned}
 \mathbb{S}_n \boxed{\mathring{c}_n} &\triangleright_m \mathbb{S}_n \boxed{\mathring{c}_n} \\
 \mathbb{S}_n \boxed{(\lambda x^n. \mathring{c}_n) t_n} &\triangleright_{\rightarrow} \mathbb{S}_n \boxed{\mathring{c}_n [t_n / x^n]} \\
 \triangleright &\stackrel{\text{def}}{=} \triangleright_m \cup \triangleright_{\rightarrow}
 \end{aligned}$$

Figure I.6.2.d: Expanded description of the operational reduction (**naive**)

$$\begin{aligned}
 \underline{t}_n v_n^1 \dots v_n^q w_n^1 \dots w_n^r &\triangleright_m \underline{t}_n v_n^1 \dots v_n^q w_n^1 \dots w_n^r \\
 (\lambda x^n. \underline{t}_n v_n^1 \dots v_n^q) w_n^0 w_n^1 \dots w_n^r &\triangleright_{\rightarrow} \underline{t}_n [w_n^0 / x^n] (v_n^1 [w_n^0 / x^n]) \dots (v_n^q [w_n^0 / x^n]) w_n^1 \dots w_n^r
 \end{aligned}$$

I. Pure call-by-name calculi

relevant in call-by-value).

The main difference with λ_N^\rightarrow is that some subterms are now also represented with commands, which allows the strong reduction \rightarrow to move focus in subterms: the expression $(\lambda x^n. I_N W_N) V_N$ of λ_N^\rightarrow can be represented in λ_n^\rightarrow by

$$\underbrace{(\lambda x^n. \underbrace{I_n w_n}) v_n}_{\text{command}} \quad \text{with} \quad I_n = \lambda y^n. \underbrace{y^n}_{\text{command}}$$

and reduces as shown in Figure I.6.1b.

Focus erasure in place of focus movement

Since subcommands now already have their own focused subterm, the reduction now erases the underlinement $\underline{\cdot}$ instead of moving it. For example, the reduction

$$\underbrace{(\lambda x^n. I_N W_N) V_N}_{\text{command}} \triangleright_m \underbrace{(\lambda x^n. I_N W_N) V_N}_{\text{command}} \triangleright_\rightarrow \underbrace{I_N W_N [V_N / x^n]}_{\text{command}}$$

becomes

$$\underbrace{(\lambda x^n. \underbrace{I_n w_n}) v_n}_{\text{command}} \triangleright_m \underbrace{(\lambda x^n. \underbrace{I_n w_n}) v_n}_{\text{command}} \triangleright_\rightarrow \underbrace{I_n w_n [v_n / x^n]}_{\text{command}}$$

where the first step erases the focus under $\underbrace{(\lambda x^n. \underbrace{I_n w_n}) v_n}_{\text{command}}$ and the second step erases the focus under $\lambda x^n. \underbrace{I_n w_n}_{\text{command}}$ and reduces the β -redex. The result $\underbrace{I_n w_n [v_n / x^n]}_{\text{command}}$ is the body $\underbrace{I_n w_n}_{\text{command}}$ of the function $\lambda x^n. \underbrace{I_n w_n}_{\text{command}}$ with the substitution $x^n \mapsto v_n$ applied to it. Erasing the focus, instead of moving it, is a way of ensuring the focus in subcommands can be moved by \rightarrow_m independently of what happens above: if we decide to first apply the reduction

$$\underbrace{(\lambda x^n. \underbrace{I_n w_n}) v_n}_{\text{command}} \rightarrow_m \underbrace{(\lambda x^n. \underbrace{I_n w_n}) v_n}_{\text{command}}$$

then the body is now $\underbrace{I_n w_n}_{\text{command}}$ and after performing the same two $\triangleright_m \triangleright_\rightarrow$ steps, we get $\underbrace{I_n w_n [v_n / x^n]}_{\text{command}}$ as expected.

I.6.2. The naive λ_n^\rightarrow calculus

Stack deferrals

We now add let-expressions to our naive λ_n^\rightarrow , which yields Figure I.6.3. The most important difference is that stacks \mathbb{S}_n can now be moved by the operational reduction \triangleright via the defer operation. Indeed, in the simple fragment, we only had simple commands for which defer only plugs the simple command in the stack:

$$\text{defer}(\underbrace{t_n \vec{v}_n}_{\text{command}}, \underbrace{\square \vec{w}_n}_{\text{stack}}) = (\underbrace{\square \vec{w}_n}_{\text{stack}}) \underbrace{t_n \vec{v}_n}_{\text{command}} = \underbrace{t_n \vec{v}_n \vec{w}_n}_{\text{command}}$$

Having let-expressions allows us to form non-simple commands for which defer moves the stack to the body of the let expression when this body is a simple command

$$\text{defer}\left(\underbrace{\text{let } x^n = \underbrace{t_n}_{\text{command}} \text{ in } \underbrace{\square \vec{w}_n}_{\text{stack}}}_{\text{command}}, \underbrace{u_n \vec{v}_n}_{\text{command}}\right) = \underbrace{\text{let } x^n = \underbrace{t_n}_{\text{command}} \text{ in } \underbrace{\square \vec{w}_n}_{\text{stack}}}_{\text{command}} = \underbrace{\text{let } x^n = \underbrace{t_n}_{\text{command}} \text{ in } \underbrace{u_n \vec{v}_n}_{\text{command}}}_{\text{command}} = \underbrace{\text{let } x^n = \underbrace{t_n}_{\text{command}} \text{ in } \underbrace{u_n \vec{v}_n \vec{w}_n}_{\text{command}}}_{\text{command}}$$

I. Pure call-by-name calculi

Figure I.6.3: The call-by-name λ -calculus with focus $\underline{\lambda}_n^{\rightarrow}$ (naive)

Figure I.6.3.a: Syntax (naive)

Expressions / values:

$$t_n, u_n, v_n, w_n ::= x^n \mid c_n \mid \lambda x^n. c_n$$

Commands:

$$c_n ::= \mathring{c}_n$$

$$| \text{ let } x^n := t_n \text{ in } c_n$$

Stacks / simple stacks:

$$S_n, \dot{S}_n ::= \square \quad | \quad S_n t_n$$

Simple commands:

$$\mathring{c}_n ::= \underbrace{t_n}_{\mathring{c}_n t_n}$$

Figure I.6.3.b: Stack deferral (naive)

$$\text{defer}(\mathring{c}_n, \mathcal{S}_n) \stackrel{\text{def}}{=} \mathcal{S}_n \boxed{\mathring{c}_n}$$
$$\text{defer}(\text{let } x^n := \underline{t_n} \text{ in } c_n, \mathbb{S}_n) \stackrel{\text{def}}{=} \text{let } x^n := \underline{t_n} \text{ in } \text{defer}(c_n, \mathbb{S}_n) \quad \text{if } x^n \text{ fresh w.r.t. } \mathbb{S}_n$$

Figure I.6.3.c: Operational reduction (naive)

$$\mathcal{S}_n \boxed{c_n} \triangleright_{\mu} \text{defer}(c_n, \mathcal{S}_n)$$
$$\text{let } x^n := \underbrace{t_n}_{\text{in}} c_n \triangleright_{\text{let}} c_n[t_n/x^n]$$
$$\mathbb{S}_n \left[\underbrace{(\lambda x^n . c_n)}_{\text{orange}} t_n \right] \triangleright_{\rightarrow} \text{defer}(c_n[t_n/x^n], \mathbb{S}_n)$$
$$\triangleright \stackrel{\text{def}}{=} \triangleright_m \cup \triangleright_{\text{let}} \cup \triangleright_{\rightarrow}$$

and further down if that body is again a let-expression

$$\text{defer} \left(\begin{array}{l} \text{let } x_1^n = \underline{t_1^n} \text{ in } \square \vec{w_n} \\ \vdots \\ \text{let } x_q^n = \underline{t_q^n} \text{ in} \\ \underline{u_n} \vec{v_n} \end{array} \right) = \dots = \text{let } x_1^n = \underline{t_1^n} \text{ in} \quad \quad \quad \text{let } x_q^n = \underline{t_q^n} \text{ in} \quad \quad \quad \text{defer}(\underline{u_n} \vec{v_n}, \square \vec{w_n})$$

 \triangleright_μ as a generalization of \triangleright_m and \boxtimes

The \triangleright_μ reduction therefore does two things: it erases the underline and moves the stack:

$$\left(\begin{array}{l} \text{let } x_1^n = \underline{t_1^n} \text{ in} \\ \vdots \\ \text{let } x_q^n = \underline{t_q^n} \text{ in} \\ \underline{u_n} \underline{v_n} \end{array} \right) \overrightarrow{w_n} \triangleright_\mu \left(\begin{array}{l} \text{let } x_1^n = \underline{t_1^n} \text{ in} \\ \vdots \\ \text{let } x_q^n = \underline{t_q^n} \text{ in} \\ \underline{u_n} \underline{v_n} \overrightarrow{w_n} \end{array} \right)$$

I. Pure call-by-name calculi

The reduction \triangleright_m of the simple fragment of $\lambda_n^{\vec{w}}$ corresponds to the case where there are $q = 0$ nested let-expressions, and the \boxtimes reduction of $\lambda_n^{\vec{w}}$ corresponds to

$$\left(\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n} \right) w_n \triangleright_\mu \text{let } x^n := \underline{t_n} \text{ in } \underline{u_n} w_n$$

i.e. to the case where there is $q = 1$ let-expression whose body is underlined (i.e. $\square \vec{v}_n = \square$) and the moved stack contains a single value (i.e. $\square \vec{w}_n = \square w_n^1$). The reduction \triangleright_μ can therefore be thought of as a combination of the reduction \triangleright_m that simply moves focus and of a strengthened variant of \boxtimes that can move several arguments at once, and move them through several let-expressions at once. This strengthening ensures that \rightarrow_μ steps commute with each other. Indeed, being able to move several values allows for

$$\begin{array}{c} \left(\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n} \right) v_n w_n \triangleright_\mu \left(\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n} \right) v_n w_n \\ \downarrow \quad \quad \quad \downarrow \\ \left(\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n} v_n \right) w_n \triangleright_\mu \text{let } x^n := \underline{t_n} \text{ in } \underline{u_n} v_n w_n \end{array}$$

and being able to move through let-expressions allows for

$$\begin{array}{c} \left(\text{let } x^n := \underline{t_n} \text{ in } \text{let } y^n := \underline{u_n} \text{ in } \underline{v_n} \right) w_n \triangleright_\mu \text{let } x^n := \underline{t_n} \text{ in } \left(\text{let } y^n := \underline{u_n} \text{ in } \underline{v_n} \right) w_n \\ \downarrow \quad \quad \quad \downarrow \\ \left(\text{let } x^n := \underline{t_n} \text{ in } \text{let } y^n := \underline{u_n} \text{ in } \underline{v_n} \right) w_n \triangleright_\mu \text{let } x^n := \underline{t_n} \text{ in } \text{let } y^n := \underline{u_n} \text{ in } \underline{v_n} w_n \end{array}$$

See [A](#) for a more formal statement.

Underlines as potential places of interaction

One way to think about the

$$\mathbb{S}_n \square \underline{c_n} \triangleright_\mu \text{defer}(c_n, \mathbb{S}_n)$$

reduction is that it moves the stack $\mathbb{S}_n = \square \vec{w}_n$ at the next point in c_n where it might interact with an expression. In non-simple commands $c_n = \text{let } x^n := \underline{t_n} \text{ in } c_n^0$, that next point of interaction of the stack with an expression is necessarily in the subcommand c_n^0 , while for simple commands $c_n = \underline{t_n} \vec{v}_n$, the stack may interact with an expression that comes from reducing $\underline{t_n} \vec{v}_n$ so we leave it here. Leaving it here does not mean that it will interact here, only that it will follow $\square \vec{v}_n$ around until all values of $\square \vec{v}_n$ have been consumed. For example, if t_n is a non-simple command, it will be moved together with $\square \vec{v}_n$:

$$\left(\text{let } x^n := \underline{u_n^1} \text{ in } \underline{u_n^2} \right) \vec{v}_n \vec{w}_n \triangleright_\mu \left(\text{let } x^n := \underline{u_n^1} \text{ in } \underline{u_n^2} \right) \vec{v}_n \vec{w}_n \triangleright_\mu \text{let } x^n := \underline{u_n^1} \text{ in } \underline{u_n^2} \vec{v}_n \vec{w}_n$$

Note that this interpretation of underlines marking points of interaction also works for let-expressions: a non-simple command

$$\text{let } x^n := \underline{t_n} \text{ in } c_n$$

is an interaction between a term t_n and an evaluation context

$$\mathcal{E}_n = \text{let } x^n := \square \text{ in } c_n$$

I. Pure call-by-name calculi

Reducing let-expressions

Note that

$\text{let } x^n := \underline{v_n} \text{ in } c_n$ and $(\lambda x^n. c_n) \underline{v_n}$ do not reduce in the same way. Indeed, while $(\lambda x^n. c_n) \underline{v_n}$ can be reduced under an arbitrary stack \mathbb{S}_n , $\text{let } x^n := \underline{t_n} \text{ in } c_n$ can not:

$$\begin{aligned} \mathbb{S}_n \left[(\lambda x^n. c_n) \underline{v_n} \right] &\triangleright_{\rightarrow} \text{defer}(c_n[t_n/x^n], \mathbb{S}_n) \quad \text{for any } \mathbb{S}_n, \text{ while} \\ \mathbb{S}_n \left[\text{let } x^n := \underline{t_n} \text{ in } c_n \right] &\triangleright_{\text{let}} \text{defer}(c_n[t_n/x^n], \mathbb{S}_n) \quad \text{only for } \mathbb{S}_n = \square \end{aligned}$$

(and for $\mathbb{S}_n \neq \square$, $\mathbb{S}_n \left[\text{let } x^n := \underline{t_n} \text{ in } c_n \right]$ is not even in the syntax). This somewhat surprising weakness of $\triangleright_{\text{let}}$ compensates for the strength of defer (and hence of the \triangleright_{μ} reduction) on let-expressions:

$$\begin{aligned} \text{defer}((\lambda x^n. c_n) \underline{v_n}, \mathbb{S}_n) &= (\lambda x^n. \text{defer}(c_n, \mathbb{S}_n)) \underline{v_n} \quad \text{only for } \mathbb{S}_n = \square, \text{ while} \\ \text{defer}(\text{let } x^n := \underline{t_n} \text{ in } c_n, \mathbb{S}_n) &= \text{let } x^n := \underline{t_n} \text{ in } \text{defer}(c_n, \mathbb{S}_n) \quad \text{for any } \mathbb{S}_n \end{aligned}$$

Indeed, the expected reduction of a let-expression under a stack can be simulated via

$$\mathbb{S}_n \left[\text{let } x^n := \underline{t_n} \text{ in } c_n \right] \triangleright_{\mu} \text{let } x^n := \underline{t_n} \text{ in } \text{defer}(c_n, \mathbb{S}_n) \triangleright_{\text{let}} \text{defer}(c_n[t_n/x^n], \mathbb{S}_n)$$

The difference between $\text{let } x^n := \underline{t_n} \text{ in } c_n$ and $(\lambda x^n. c_n) \underline{v_n}$ is therefore that defer understands that $\text{let } x^n := \underline{t_n} \text{ in } c_n$ will reduce without interacting with the surrounding stack, but has no such knowledge for $(\lambda x^n. c_n) \underline{v_n}$ ⁵. This implies that $\mathbb{S}_n \left[\text{let } x^n := \underline{t_n} \text{ in } c_n \right]$ is \triangleright_{μ} reducible, and since we want \triangleright to be deterministic (and \triangleright_{l_1} and \triangleright_{l_2} to have disjoint domains for $l_1 \neq l_2$), it can not be $\triangleright_{\text{let}}$ -reducible, which is why $\triangleright_{\text{let}}$ is weaker than expected.

Undesirable strong reductions

In this naive version of λ_n^{\rightarrow} , the strong reduction \rightarrow^6 is somewhat unsatisfying because we do not have

$$\left. \begin{array}{l} c_n \triangleright c'_n, \text{ and} \\ \mathbb{K}_n \left[c_n \right] \text{ is in the syntax} \end{array} \right\} \Rightarrow \mathbb{K}_n \left[c_n \right] \rightarrow \mathbb{K}_n \left[c'_n \right]$$

but only

$$\left. \begin{array}{l} c_n \triangleright c'_n, \\ \mathbb{K}_n \left[c_n \right] \text{ is in the syntax, and} \\ \mathbb{K}_n \left[c'_n \right] \text{ is in the syntax} \end{array} \right\} \Rightarrow \mathbb{K}_n \left[c_n \right] \rightarrow \mathbb{K}_n \left[c'_n \right]$$

⁵Note that \triangleright_{μ} sometimes failing to recognize that a term will not interact with the surrounding stack is perfectly reasonable: “interacting with the surrounding stack” is an undecidable property (e.g. because a closed term interacts with the surrounding stack if and only if it terminates), so that \triangleright_{μ} is necessarily an approximation.

⁶The strong reduction \rightarrow can be defined either by

$$t \rightarrow t' \stackrel{\text{def}}{=} \exists \mathbb{K}_n, \exists c_n, \exists c'_n, \left\{ \begin{array}{l} c_n \triangleright c'_n, \\ t = \mathbb{K}_n \left[c_n \right], \text{ and} \\ t' = \mathbb{K}_n \left[c'_n \right] \end{array} \right.$$

or equivalently by the expected inference rules.

I. Pure call-by-name calculi

For example, the reduction

$$c_n = (\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n})v_n \rightarrow_\mu \text{let } x^n := \underline{t_n} \text{ in } \underline{u_n}v_n = c'_n$$

is not preserved under $\mathbb{k}_n = \square w_n$ because $(\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n})v_n$ is not in the syntax:

$$\begin{array}{ccc} \mathbb{k}_n \boxed{c_n} & & \mathbb{k}_n \boxed{c'_n} \\ \parallel & & \parallel \\ (\square w_n) \boxed{(\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n})v_n} & & (\square w_n) \boxed{\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n}v_n} \\ \parallel & & \parallel \\ (\underline{\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n}})v_n w_n & \not\rightarrow_\mu & (\text{let } x^n := \underline{t_n} \text{ in } \underline{u_n}v_n)w_n \end{array}$$

This is due to a reduction

$$\mathbb{s}_n^2 \boxed{\mathbb{s}_n^1 \boxed{c_n}} \rightarrow_\mu \mathbb{s}_n^2 \boxed{\text{defer}(c_n, \mathbb{s}_n^1)}$$

only being valid when there is no potential interaction between $\text{defer}(c_n, \mathbb{s}_n^1)$ and \mathbb{s}_n^2 , because if there is, the syntax requires making it with $\underline{\cdot}$, i.e. writing $\mathbb{s}_n^2 \boxed{\text{defer}(c_n, \mathbb{s}_n^1)}$. In other words, since \rightarrow_μ erases $\underline{\cdot}$, it must ensure that there is no potential interaction at that $\underline{\cdot}$ by deferring the whole stack $\mathbb{s}_n^2 \mathbb{s}_n^1$.

This can also be understood as stemming from the fact that simple commands are not stable under \triangleright_μ : $\mathbb{s}_n^1 \boxed{c_n}$ is a simple command and can hence be plugged in \mathbb{s}_n^2 while remaining within the syntax, while $\text{defer}(c_n, \mathbb{s}_n^1)$ is simple only if c_n is, and $\mathbb{s}_n^2 \boxed{\text{defer}(c_n, \mathbb{s}_n^1)}$ is therefore in the syntax only if $\mathbb{s}_n^2 = \square$ or c_n is simple.

I.6.3. The λ_n^\rightarrow calculus

Explicit command boundaries

The pure untyped call-by-name λ -calculus with focus λ_n^\rightarrow is defined in Figure I.6.4. It deals with the aforementioned quirks of the strong reduction by explicitly marking the top of commands with a constructor com^n , i.e. replacing

$$c_n ::= \mathring{c}_n \quad \text{by} \quad c_n ::= \text{com}^n(\mathring{c}_n),$$

and restricting \triangleright by only allowing it to reduce objects that have com^n above them. This prevents the problematic \rightarrow_μ reductions because there is no com^n around $\mathbb{s}_n^1 \boxed{c_n}$ in $\text{com}^n(\mathbb{s}_n^2 \mathbb{s}_n^1 \boxed{c_n})$, and it can therefore not be reduced on its own. Adding the com^n yields an invalid term $\text{com}^n(\mathbb{s}_n^2 \boxed{\text{com}^n(\mathbb{s}_n^1 \boxed{c_n})})$, unless we also add a $\underline{\cdot}$, which yields $\text{com}^n(\mathbb{s}_n^2 \boxed{\underline{\text{com}^n(\mathbb{s}_n^1 \boxed{c_n})}})$ for which the problematic is disallowed by the extra $\underline{\cdot}$ (and com^n).

The need for com^n was to be expected. Indeed, the calculus λ_n^\rightarrow was built as an outside-out representation of Li_n^\rightarrow (defined in the next section), and commands $\langle t_n | e_n \rangle$ of Li_n^\rightarrow have both an explicit marker $|$ for the point of potential interaction between t_n and e_n ; and an explicit marker $\langle \cdot \rangle$ for the top of the command. It therefore makes sense for commands $\text{com}^n(\mathbb{s}_n \boxed{t_n})$ of λ_n^\rightarrow to have both an explicit marker $\underline{\cdot}$ for the point of interaction between \mathbb{s}_n and t_n ; and an explicit marker com^n for the top of the command.

Figure I.6.4: The pure call-by-name λ -calculus with focus $\underline{\lambda}_n^{\vec{\lambda}}$

Figure I.6.4.a: Syntax

Expressions / values:

$$t_n, u_n, v_n, w_n ::= x^n \mid \text{ctot}^n(c_n) \mid \lambda x^n. c_n$$

Commands:

$$c_n ::= \text{com}^n(\dot{c}_n) \mid \text{com}^n(\text{let } x^n := \underline{t}_n \text{ in } c_n)$$

Stacks / simple stacks:

$$\mathbb{S}_n, \dot{\mathbb{S}}_n ::= \square \mid \mathbb{S}_n t_n$$

Incomplete simple commands:

$$\dot{c}_n ::= \text{instk}^n(\underline{t}_n) \mid \dot{c}_n t_n$$

Figure I.6.4.b: Notations

Evaluation contexts:

$$\mathcal{E}_n ::= \mathbb{S}_n [\text{instk}^n(\square)] \mid \text{let } x^n := \square \text{ in } c_n$$

Simple commands:

$$c_n^{\text{simple}} ::= \text{com}^n(\dot{c}_n)$$

Terms:

$$t ::= t_n \mid \dot{c}_n \mid c_n$$

Figure I.6.4.c: Deferred stacks

$$\begin{aligned} \text{defer}(\dot{c}_n, \mathbb{S}_n) &\stackrel{\text{def}}{=} \mathbb{S}_n [\dot{c}_n] \\ \text{defer}(\text{let } x^n := \underline{t}_n \text{ in } c_n, \mathbb{S}_n) &\stackrel{\text{def}}{=} \text{let } x^n := \underline{t}_n \text{ in } \text{defer}(c_n, \mathbb{S}_n) \end{aligned}$$

Figure I.6.4.d: Deferred stacks (in evaluation contexts)

$$\begin{aligned} \text{defer}(\dot{\mathbb{S}}_n, \mathbb{S}_n) &\stackrel{\text{def}}{=} \mathbb{S}_n [\dot{\mathbb{S}}_n] \\ \text{defer}(\text{let } x^n := \square \text{ in } c_n, \mathbb{S}_n) &\stackrel{\text{def}}{=} \text{let } x^n := \square \text{ in } \text{defer}(c_n, \mathbb{S}_n) \end{aligned}$$

Figure I.6.4.e: Operational reduction

$$\begin{aligned} \text{com}^n(\mathbb{S}_n [\underline{c}_n]) &\triangleright_{\mu} \text{defer}(c_n, \mathbb{S}_n) \\ \text{com}^n(\text{let } x^n := \underline{t}_n \text{ in } c_n) &\triangleright_{\text{let}} c_n[t_n/x^n] \\ \text{com}^n(\mathbb{S}_n [\underline{(\lambda x^n. c_n) t_n}]) &\triangleright_{\rightarrow} \text{defer}(c_n[t_n/x^n], \mathbb{S}_n) \\ \triangleright &\stackrel{\text{def}}{=} \triangleright_{\mu} \cup \triangleright_{\text{let}} \cup \triangleright_{\rightarrow} \end{aligned}$$

 Figure I.6.4.f: Top-level η -expansion

$$\begin{aligned} t_n &\stackrel{\eta}{\delta_{\mu}} \text{ctot}^n(\underline{t}_n) \\ \mathcal{E}_n &\stackrel{\eta}{\delta_{\text{let}}} \text{let } x^n := \square \text{ in } \mathcal{E}_n [\underline{x^n}] \quad \text{if } x^n \text{ fresh w.r.t. } \mathcal{E}_n \\ t_n &\stackrel{\eta}{\delta_{\rightarrow}} \lambda x^n. \underline{t_n x^n} \quad \text{if } x^n \text{ fresh w.r.t. } t_n \\ \delta &\stackrel{\eta}{\text{def}} \stackrel{\eta}{\delta_{\mu}} \cup \stackrel{\eta}{\delta_{\text{let}}} \cup \stackrel{\eta}{\delta_{\rightarrow}} \end{aligned}$$

Coercions

In addition to explicit command boundaries com^n , λ_n^\rightarrow has an explicit coercion ctot^n from command c_n to expressions t_n (hence the name ctot) and an explicit marker instk^n around underlines $\underline{\cdot}$ that will be placed within a stack \mathbb{S}_n . Both com^n and instk^n are often left implicit because the former is only relevant when defining \rightarrow , and the latter is only relevant when studying translations between λ_n^\rightarrow and Li_n^\rightarrow . In particular, we will often denote simple commands by \hat{c}_n instead of $\text{com}^n(\hat{c}_n)$. While ctot^n could also often be left implicit, not distinguishing the command c_n from the expression $\text{ctot}^n(c_n)$ may lead to some confusion, and we therefore keep ctot^n explicit for the sake of clarity.

Evaluation contexts

Evaluation contexts \mathcal{E}_n form a superset of stacks that are not required to define λ_n^\rightarrow . Commands are exactly terms of the shape $\text{com}^n(\mathcal{E}_n \underline{t_n})$ (see [A](#)), and evaluation contexts are therefore useful whenever this inside-out description of commands is, e.g. when studying at translations between λ_n^\rightarrow and Li_n^\rightarrow .

The difference between a stack and an evaluation context is that an a stack \mathbb{S}_n can be deferred so that something else is computed first, while a non-stack evaluation context \mathcal{E}_n can not. For example, placing $t_n = \text{let } x^n := \underline{v_n} \text{ in } \underline{x^n}$ under $\mathbb{S}_n = \square w_n$ results in \mathbb{S}_n being moved

$$\mathbb{S}_n \underline{t_n} = (\text{let } x^n := \underline{v_n} \text{ in } \underline{x^n}) w_n \triangleright_\mu \text{let } x^n := \underline{v_n} \text{ in } \underline{x^n} w_n = \text{let } x^n := \underline{v_n} \text{ in } \mathbb{S}_n \underline{x^n}$$

while placing it inside $\mathcal{E}_n = \text{let } y^n := \square \text{ in } c_n$ results in this let-expressiong being evaluated

$$\mathcal{E}_n \underline{t_n} = \text{let } y^n := (\text{let } x^n := \underline{v_n} \text{ in } \underline{x^n}) \text{ in } c_n \triangleright_{\text{let}} c_n [\text{let } x^n := \underline{v_n} \text{ in } \underline{x^n} / y^n] = c_n [t_n / y^n]$$

Disubstitution

Disubstitutions of λ_n^\rightarrow are defined just like in λ_n^\rightarrow , with plugging replaced by defer:

Definition I.6.1

A *disubstitution* φ is a pair $\varphi = (\sigma, \mathbb{S}_n)$ composed of a substitution σ and a stack \mathbb{S}_n . The action of a disubstitution $\varphi = (\sigma, \mathbb{S}_n)$ on commands (resp. evaluations contexts) is defined by

$$c_n[\varphi] \stackrel{\text{def}}{=} \text{defer}(c_n[\sigma], \mathbb{S}_n) \quad (\text{resp. } \mathcal{E}_n \stackrel{\text{def}}{=} \text{defer}(\mathcal{E}_n, \mathbb{S}_n))$$

and its action on expressions^a is defined by

$$t_n[\varphi] \stackrel{\text{def}}{=} t_n[\sigma]$$

The composition $\varphi_1[\varphi_2]$ of two disubstitutions is defined by

$$(\sigma_1, \mathbb{S}_n^1)[\sigma_2, \mathbb{S}_n^2] \stackrel{\text{def}}{=} (\sigma_1[\sigma_2], \mathbb{S}_n^1[\sigma_2])$$

^aSee [A](#).

I. Pure call-by-name calculi

Fact I.6.2

The set of disubstitutions φ_n has a monoid structure

$$(\varphi_n, \circ, (\text{Id}_V, \square)) \quad \text{where} \quad \varphi_2 \circ \varphi_1 \stackrel{\text{def}}{=} \varphi_1[\varphi_2]$$

and this monoid acts on commands, expressions, and evaluation contexts via

$$\varphi \bullet t \stackrel{\text{def}}{=} t[\varphi]$$

In particular, defer is associative: for any command c_n and stacks \mathbb{S}_n^1 and \mathbb{S}_n^2 , we have

$$\text{defer}(\text{defer}(c_n, \mathbb{S}_n^1), \mathbb{S}_n^2) = \text{defer}(c_n, \text{defer}(\mathbb{S}_n^1, \mathbb{S}_n^2))$$

Proof



Reductions

We write \rightarrow for the contextual closure of \triangleright , \rightarrow for the contextual closure of \triangleright , $\vdash \triangleright$ for $\rightarrow \cup \vdash$ and $\langle \vdash \triangleright \rangle$ for $\vdash \triangleright \cup \langle \vdash \triangleright \rangle$ (i.e. $\rightarrow \cup \leftarrow \cup \rightarrow \cup \vdash$). The η -expansion for functions is the usual one with a conversion $\underline{}$ from expressions to commands added, and the other η -expansions are easier to understand in the $\text{Li}_n^{\rightarrow}$ where they look natural, and can safely be ignored for now. The reductions have the properties announced in Figure ?? (see Section .2 for details).

I.7. Translations between $\lambda_{\mathbf{N}}^{\rightarrow}$ and $\lambda_{\mathbf{n}}^{\rightarrow}$



I.8. A pure call-by-name intuitionistic L calculus: Li_n^\rightarrow

In this section, we recall the intuitionistic call-by-name fragments of $\bar{\lambda}\mu\tilde{\mu}$ [CurHer00], which we call Li_n^\rightarrow .

I.8.1. From the M_N^\rightarrow abstract machine to the Li_n^\rightarrow calculus

Decomposing the strong reduction

Just like in $\underline{\lambda}_N^\rightarrow$, the strong reduction \rightarrow is unsatisfying in M_N^\rightarrow because it can not be decomposed like \triangleright can, as shown in Figure I.8.1a. The naive Li_n^\rightarrow calculus defined in Figure I.8.2 fixes this, as shown in Figure I.8.1b (where $I_n = \lambda y^n. \langle y^n | \star^n \rangle$), in the same way that $\underline{\lambda}_N^\rightarrow$ did: by representing the body of λ -abstraction by configurations / commands.

Pattern matching stacks

In the actual simple fragment of the Li_n^\rightarrow calculus described in Figure I.8.3, the stack \star^n is thought of as being a stack variable, and λ -abstractions $\lambda x^n. c_n$ are denoted by $\mu(x^n \cdot \star^n). c_n$ to emphasize that \star^n is bound in $\mu(x^n \cdot \star^n). c_n$, and hence that the disubstitution $\star^n \mapsto s_n$ acts trivially on $\mu(x^n \cdot \star^n). c_n$:

$$(\mu(x^n \cdot \star^n). c_n)[s_n / \star^n] = \mu(x^n \cdot \star^n). c_n$$

This allows for a more succinct description of the $\triangleright_{\rightarrow}$ reduction

$$\langle \mu(x^n \cdot \star^n). c_n | v_n \cdot s_n \rangle \triangleright_{\rightarrow} c_n[v_n / x^n, s_n / \star^n]$$

which can be thought of as pattern-matching the stack $v_n \cdot s_n$ against the stack pattern $x^n \cdot \star^n$ and applying the unifier $x^n \mapsto v_n, \star^n \mapsto s_n$ to the command c_n .

Stack variable names

Note that just like the same name x can be used for several unrelated value variables x^n in the same expression, the name \star can be used for several unrelated stack variables. For example,

$$I_n I_n = (\lambda x^n. \underline{x^n})(\lambda x^n. \underline{x^n})$$

stands for

$$(\lambda x_1^n. \underline{x_1^n})(\lambda x_2^n. \underline{x_2^n})$$

and similarly

$$\langle I_n | I_n \cdot \star^n \rangle = \langle \mu(x^n \cdot \star^n). \langle x^n | \star^n \rangle | (\mu(x^n \cdot \star^n). \langle x^n | \star^n \rangle) \cdot \star^n \rangle$$

stands for

$$\langle \mu(x_1^n \cdot \star_1^n). \langle x_1^n | \star_1^n \rangle | (\mu(x_2^n \cdot \star_2^n). \langle x_2^n | \star_2^n \rangle) \cdot \star_0^n \rangle$$

The difference is that we have infinitely many value variables available, and can therefore always rename the bound ones to avoid such name clashes, but only one stack variable in Li_n^\rightarrow and can therefore not avoid such clashes. The stack variable \star^n can alternatively be thought of as being the 0 de Bruijn index for stack variables. The difference between the

[CurHer00] “The duality of computation”, Curien and Herbelin, 2000

Figure I.8.1: Example of strong reduction in subterms of abstract machines

Figure I.8.1.a: Example in M_N^{\rightarrow}

$$\begin{array}{ccc}
\langle (\lambda x^N. I_N W_N V_N | \star^N) \rangle_{\mathfrak{m}} & \langle \lambda x^N. I_N W_N | V_N \cdot \star^N \rangle & \langle I_N W_N [V_N / x^N] | \star^N \rangle \\
\downarrow & \downarrow & \Downarrow \\
\langle (\lambda x^n. \langle w_n | \star^n \rangle) v_n | \star^n \rangle & \langle \lambda x^n. \langle w_n | \star^n \rangle | v_n \cdot \star^n \rangle & \langle I_N | W_N [V_N / x^N] \cdot \star^N \rangle \\
\downarrow & \downarrow & \downarrow \\
\langle (\lambda x^n. \langle w_n | \star^n \rangle) v_n | \star^n \rangle & \langle \lambda x^n. \langle w_n | \star^n \rangle | v_n \cdot \star^n \rangle & \langle W_N [V_N / x^N] | \star^N \rangle
\end{array}$$

Figure I.8.1.b: Example in naive $\text{Li}_n^{\rightarrow}$

$$\begin{array}{ccccc}
\langle \lambda x^n. \langle I_n w_n | \star^n \rangle v_n | \star^n \rangle & \triangleright_m & \langle \lambda x^n. \langle I_n w_n | \star^n \rangle | v_n \cdot \star^n \rangle & \triangleright_{\rightarrow} & \langle I_n w_n [v_n / x^n] | \star \rangle \\
\downarrow \exists & & \downarrow \exists & & \downarrow \exists \\
\langle (\lambda x^n. \langle I_n | w_n \cdot \star^n \rangle) v_n | \star^n \rangle & \triangleright_m & \langle \lambda x^n. \langle I_n | w_n \cdot \star^n \rangle | v_n \cdot \star^n \rangle & \triangleright_{\rightarrow} & \langle I_n | w_n [v_n / x^n] \cdot \star^n \rangle \\
\downarrow & & \downarrow & & \downarrow \\
\langle (\lambda x^n. \langle w_n | \star^n \rangle) v_n | \star^n \rangle & \triangleright_m & \langle \lambda x^n. \langle w_n | \star^n \rangle | v_n \cdot \star^n \rangle & \triangleright_{\rightarrow} & \langle w_n [v_n / x^n] | \star^n \rangle
\end{array}$$

Figure I.8.1.c: Example in Li_n^+

$$\begin{array}{ccccc}
\langle \mu(x^n \cdot \star^n). \langle I_n w_n | \star^n \rangle \rangle v_n | \star^n & \triangleright_\mu & \langle \mu(x^n \cdot \star^n). \langle I_n w_n | \star^n \rangle | v_n \cdot \star^n \rangle & \triangleright_\rightarrow & \langle I_n w_n [v_n / x^n] | \star \rangle \\
\downarrow \cong & & \downarrow \cong & & \downarrow \cong \\
\langle (\mu(x^n \cdot \star^n). \langle I_n | w_n \cdot \star^n \rangle) v_n | \star^n \rangle & \triangleright_\mu & \langle \mu(x^n \cdot \star^n). \langle I_n | w_n \cdot \star^n \rangle | v_n \cdot \star^n \rangle & \triangleright_\rightarrow & \langle I_n | w_n [v_n / x^n] \cdot \star^n \rangle \\
\downarrow & & \downarrow & & \downarrow \\
\langle (\mu(x^n \cdot \star^n). \langle w_n | \star^n \rangle) v_n | \star^n \rangle & \triangleright_\mu & \langle \mu(x^n \cdot \star^n). \langle w_n | \star^n \rangle | v_n \cdot \star^n \rangle & \triangleright_\rightarrow & \langle w_n [v_n / x^n] | \star^n \rangle
\end{array}$$

two interpretations is only relevant when looking at the inclusion of Li_n^\rightarrow into L_n^\rightarrow , and will be discussed in Section I.10.

Binding the stack variable

Since we think of \star^n as a variable, one can add a binder $\mu\star^n.c_n$ for it, with the reduction

$$\langle \mu \star^n . c_n | S_n \rangle \triangleright_\mu c_n[S_n / \star^n]$$

and define $t_n u_n$ as a notation for $\mu \star^n \langle t_n | u_n \cdot \star^n \rangle$. Indeed, with this notation, the reduction

$$\langle t_n u_n | s_n \rangle \triangleright_m \langle t_n | u_n \cdot s_n \rangle$$

I. Pure call-by-name calculi

Figure I.8.2: The simple fragment of the **naive** $\text{Li}_n^{\rightarrow}$ calculus

Figure I.8.2.a: Syntax (**naive**)

<p>Terms / values:</p> $t_n, u_n, v_n, w_n ::= x^n \mid t_n u_n$ $\mid \lambda x^n. c_n$ <p>Commands:</p> $c_n ::= \langle t_n \mid s_n \rangle$	<p>Stacks:</p> $s_n ::= \star^n$ $\mid t_n \cdot s_n$
--	---

Figure I.8.2.b: Operational reduction (**naive**)

$$\begin{aligned}
 & \langle t_n u_n \mid s_n \rangle \triangleright_m \langle t_n \mid u_n \cdot s_n \rangle \\
 & \langle \lambda x^n. \langle t_n \mid \vec{w}_n \cdot \star^n \rangle \mid v_n^0 \cdot \vec{v}_n \cdot \star^n \rangle \triangleright_{\rightarrow} \langle t_n \mid v_n^0 / x^n \rangle \mid \vec{w}_n[v_n^0 / x^n] \cdot \vec{v}_n \cdot \star^n \rangle \\
 & \triangleright \stackrel{\text{def}}{=} \triangleright_m \cup \triangleright_{\rightarrow}
 \end{aligned}$$

Figure I.8.3: The simple fragment of the $\text{Li}_n^{\rightarrow}$ calculus

Figure I.8.3.a: Syntax

<p>Terms / values:</p> $t_n, u_n, v_n, w_n ::= x^n \mid \mu \star^n. c_n$ $\mid \mu(x^n \cdot \star^n). c_n$ <p>Commands:</p> $c_n ::= \langle t_n \mid s_n \rangle$	<p>Stacks:</p> $s_n ::= \star^n$ $\mid t_n \cdot s_n$
--	---

Figure I.8.3.b: Operational reduction

$$\begin{aligned}
 & \langle \mu \star^n. c_n \mid s_n \rangle \triangleright_{\mu} c_n[s_n / \star^n] \\
 & \langle \mu(x^n \cdot \star^n). c_n \mid v_n \cdot s_n \rangle \triangleright_{\rightarrow} c_n[v_n / x^n, s_n / \star^n] \\
 & \triangleright \stackrel{\text{def}}{=} \triangleright_{\mu} \cup \triangleright_{\rightarrow}
 \end{aligned}$$

is a special case of \triangleright_{μ} :

$$\langle t_n u_n \mid s_n \rangle = \langle \mu \star^n. \langle t_n \mid u_n \cdot \star^n \rangle \mid s_n \rangle \triangleright_{\mu} \langle t_n \mid u_n \cdot s_n \rangle$$

Remark I.8.1

For this particular calculus, removing $t_n u_n$ from the syntax and adding $\mu\star^n.c_n$ does not change the calculus much. However, in L calculi with more constructors, more expressions can be expressed with $\mu\star^n.c_n$, so that the calculus with $\mu\star^n.c_n$ ends up being simpler. More precisely, without $\mu\star^n.c_n$, every stack constructor (e.g. $u_n \cdot s_n$) needs to have an associated expression constructor (e.g. $t_n u_n$), while with $\mu\star^n.c_n$, the expression constructors can be defined as notation (e.g. $t_n u_n \stackrel{\text{ntn}}{=} \mu\star^n.\langle t_n | s_n \cdot \star^n \rangle$). The gain is therefore linear in the number of stack constructors. Here, nothing is gained because there is only one stack constructor, but for larger calculi such as those in Chapter IV, the gain is non-negligible.

I.8.2. The $\text{Li}_n^{\vec{}}$ calculus

Let-expressions and $\tilde{\mu}$

The full $\text{Li}_n^{\vec{}}$ calculus, described in Figure I.8.4, is its simple fragment extended by commands $\langle t_n | \tilde{\mu}x^n.c_n \rangle$ that represent let-expressions $\text{let } x^n := t_n \text{ in } c_n$ (and $\tilde{\mu}x^n.c_n$ that represents $\text{let } x^n := \square \text{ in } c_n$). Their reduction is exactly what one would expect:

$$\langle t_n | \tilde{\mu}x^n.c_n \rangle \triangleright_{\tilde{\mu}} c_n[t_n/x^n]$$

Note that commands could be defined without referring to evaluation contexts e_n :


$$c_n ::= \langle t_n | s_n \rangle \mid \langle t_n | \tilde{\mu}x^n.c_n \rangle$$

Indeed, $\tilde{\mu}x^n.c_n$ can only appear inside contexts of the shape $\langle t_n | \square \rangle$. It is nevertheless useful to have $\tilde{\mu}x^n.c_n$ be a term on its own because it makes the calculus more symmetric and makes expressing some definitions nicer (e.g. η -expansion).

Coercions



Disubstitutions

Disubstitutions have the properties announced in  (see Section .1 for details).

Reductions

The reductions have the properties announced in Figure ?? (see Section .2 for details).

I. Pure call-by-name calculi

Figure I.8.4: The Li_n^\rightarrow calculus

Figure I.8.4.a: Syntax

Terms / values:	Stacks:
$t_n, u_n, v_n, w_n ::= x^n \mid \mu \star^n . c_n$	$s_n ::= \star^n$
$\mid \mu(x^n \cdot \star^n) . c_n$	$\mid t_n \cdot s_n$
Commands:	Evaluation contexts:
$c_n ::= \langle t_n \mid e_n \rangle$	$e_n ::= \text{stk}^n(s_n) \mid \tilde{\mu} x^n . c_n$

Figure I.8.4.b: Notations

$$t_n u_n \stackrel{\text{ntn}}{=} \mu \star^n . \langle t_n \mid u_n \cdot \star^n \rangle$$

Figure I.8.4.c: Operational reduction

$$\begin{aligned}
&\langle \mu \star^n . c_n \mid s_n \rangle \triangleright_\mu c_n[s_n / \star^n] \\
&\langle t_n \mid \tilde{\mu} x^n . c_n \rangle \triangleright_{\tilde{\mu}} c_n[t_n / x^n] \\
&\langle \mu(x^n \cdot \star^n) . c_n \mid v_n \cdot s_n \rangle \triangleright_{\rightarrow} c_n[v_n / x^n, s_n / \star^n] \\
&\triangleright \stackrel{\text{def}}{=} \triangleright_\mu \cup \triangleright_{\tilde{\mu}} \cup \triangleright_{\rightarrow}
\end{aligned}$$

Figure I.8.4.d: η -expansion

$$\begin{aligned}
t_n &\Downarrow_\mu \mu \star^n . \langle t_n \mid \star^n \rangle \\
e_n &\Downarrow_{\text{let}} \tilde{\mu} x^n . \langle x^n \mid e_n \rangle && \text{if } x^n \text{ fresh w.r.t. } e_n \\
t_n &\Downarrow_{\rightarrow} \mu(x^n \cdot \star^n) . \langle t_n \mid x^n \cdot \star^n \rangle && \text{if } x^n \text{ fresh w.r.t. } t_n \\
\Downarrow &\stackrel{\text{def}}{=} \Downarrow_\mu \cup \Downarrow_{\tilde{\mu}} \cup \Downarrow_{\rightarrow}
\end{aligned}$$

I.9. Equivalence between λ_n^\rightarrow and Li_n^\rightarrow



I. Pure call-by-name calculi

I.10. A pure call-by-name classical L calculus: L_n^{\rightarrow}



I.11. Simply-typed λ calculi



II. Pure call-by-value calculi



II. Pure call-by-value calculi

II.1. A pure call-by-value λ -calculus: $\lambda_{\text{V}}^{\rightarrow}$



II. Pure call-by-value calculi

II.2. A pure call-by-value λ -calculus with focus: $\lambda_{\underline{v}}^{\rightarrow}$



II. Pure call-by-value calculi

II.3. A pure call-by-value intuitionistic L calculus: Li_v^\rightarrow



II. Pure call-by-value calculi


II.4. A pure call-by-value classical L calculus: L_v^{\rightarrow}



Part B.

Untyped polarized calculi

Introduction

Introduction  The λ -calculus [Bar84] is a well-known abstraction used to study programming languages and has two main evaluation strategies: *call-by-name* (CBN) and *call-by-value* (CBV). A study of their typed models lead to *Call-by-push-value* (CBPV) [Lev04; Lev06], a calculus that decomposes and subsumes both CBV and CBN. For quite some time, the study of CBPV was largely focused on its (denotational) semantics in a typed setting, but recently the rewriting theory of the pure implicative fragment of untyped CBPV has been studied in an alternative syntax called the Bang calculus [Ehr16; EhrGue16; BucKesRío-Vis20].

One of the main difficulties is that, just like in call-by-value [AccGue16], the usual way of defining the operational reduction yields an ill-behaved reduction: there are expressions that should diverge but are reducible by neither the operational reduction nor its contextual closure¹. Neither of the two solutions to this problem that were put forward in the Bang calculus seem to scale well to all of CBPV². The $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ -calculus (also called³ LJ_p^η [CurFioMun16] or L_{int} [MunSch15]) also solves this problem thanks to the μ -reductions that plays the same role as the σ -reductions in [EhrGue16], but can be described far more succinctly. The $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ -calculus is related to the $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ -calculus, our “completed” variant of CBPV, and hence also to CBPV.

Another difficulty is the presence of clashes, i.e. interactions between constructors and / or destructors that were not means to interact. Clashes between positive and negative parts of the calculus can be removed simply by restricting the syntax in a way that loses no expressivity, and are already absent from $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$. Clashes between two positive parts or two negative parts however are inherent to the $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ -calculus. In many well-known dynamically-typed programming languages, it is possible to write pattern-matches that match over constructors of several types (by first matching on the type, and then on the constructors within that type). This super-pattern-match can handle any positive expression (i.e. anything except a function in these programming languages) without generating a clash (i.e. a runtime type error). By dualizing this idea in $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$, we get negative expressions that can han-

¹For example, $T = (\text{let } x := yI \text{ in } \lambda _ . \delta) \delta$ with $\delta = \lambda z. zz$ is both \triangleright -normal and \dashvtriangleright -normal, but $T[\sigma] \triangleright$ -diverges for any closed substitution σ .

²The number of necessary σ -reductions grows quadratically in the size of the calculus, and the “reduction at a distance” paradigm seems to break when trying to generalize it to sums.

³Up to minor differences.

[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

[Lev04] *Call-By-Push-Value: A Functional/Imperative Synthesis*, Levy, 2004

[Lev06] “Call-by-push-value: Decomposing call-by-value and call-by-name”, Levy, 2006

[Ehr16] “Call-By-Push-Value from a Linear Logic Point of View”, Ehrhard, 2016

[EhrGue16] “The Bang Calculus: An Untyped Lambda-Calculus Generalizing Call-by-Name and Call-by-Value”, Ehrhard and Guerrieri, 2016

[BucKesRíoVis20] “The Bang Calculus Revisited”, Bucciarelli *et al.*, 2020

[AccGue16] “Open Call-by-Value”, Accattoli and Guerrieri, 2016

[CurFioMun16] “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”, Curien, Fiore, and Munch-Maccagnoni, 2016

[MunSch15] “Polarised Intermediate Representation of Lambda Calculus with Sums”, Munch-Maccagnoni and Scherer, 2015

dle any negative stack⁴. By replacing all pattern-matches and negative expressions by this super-pattern-match and its dual, we get a calculus $\text{Li}_p^{\mathcal{PN}}$ with no clashes in which $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ can be embedded. While $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ and $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ are clearly the right calculi in a typed setting, there is a case to be made for $\lambda_p^{\mathcal{PN}}$ and $\text{Li}_p^{\mathcal{PN}}$ being the right calculi in the untyped setting. We contribute a first piece of evidence to that case in Part C by showing that there is a (relatively) simple operational characterization of solvability in $\text{Li}_p^{\mathcal{PN}}$, but that any such characterization in $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ would be fairly complex.

Content 

Contribution 

⁴In $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$, this implies having expressions T_- that can act both as a function $\lambda x^+. U_-^1$ and as a negative pair $(U_-^2 \& U_-^3)$, i.e. such that $T_- V_+ \triangleright U_-^1[V_+/x^+]$ and $\pi_i(T_-) \triangleright U_-^{i+1}$.

III. Pure polarized calculi

III.1. Relative expressiveness of call-by-name and call-by-value

The fundamental distinction between call-by-name and call-by-value is how let-expressions are reduced. In call-by-name a let-expression $\text{let } x^N := T_N \text{ in } U_N$ is immediately reduced to $U_N[T_N/x^N]$ (i.e. any T_N is considered to be a value V_N), whereas in call-by-value the expression T_V is first reduced until it reaches a value W_V (and if it never does, i.e. T_V diverges, then so does $\text{let } x^V := T_V \text{ in } U_V$) and only then does the substitution happen.

$$\begin{aligned} \text{let } x^N := T_N \text{ in } U_N &= \text{let } x^N := V_N \text{ in } U_N \triangleright U_N[V_N/x^N] \\ \text{let } x^V := T_V \text{ in } U_V &\triangleright^* \text{let } x^V := W_V \text{ in } U_V \triangleright U_V[W_V/x^V] \end{aligned}$$

With that in mind, we now look at how λ_N^\rightarrow and λ_V^\rightarrow can be embedded in each other in direct style (i.e. not in continuation-passing style). In Section III.1.1, we give an embedding of λ_N^\rightarrow into a slight extension of λ_V^\rightarrow called $\lambda_V^{\rightarrow\uparrow}$ and in Section III.1.2, we give an embedding of λ_V^\rightarrow into a slight extension of λ_N^\rightarrow called $\lambda_N^{\rightarrow\downarrow}$. Since there is a translation from $\lambda_V^{\rightarrow\uparrow}$ to λ_V^\rightarrow , we could have embedded λ_N^\rightarrow into λ_V^\rightarrow directly, but introducing $\lambda_V^{\rightarrow\uparrow}$ makes the translation easier to understand, and the similarity between both translation more apparent.

The goal of this section is only to give some intuition in what shifts represent, and not to prove any formal result. In Section III.2, formal results are given for similar translations from λ_N^\rightarrow and λ_V^\rightarrow to $\lambda_P^{\rightarrow\uparrow\downarrow}$ that refine the translations of this section.

III.1.1. Embedding call-by-name in call-by-value

The $\lambda_V^{\rightarrow\uparrow}$ -calculus

The extension of λ_V^\rightarrow , called $\lambda_V^{\rightarrow\uparrow}$, is defined in Figure III.1.1. Given a computation T_V , we add $\text{freeze}^V(T_V)$ which represents the computation T_V paused:

$$\text{freeze}^V(T_V) \triangleright$$

The computation can later be resumed with unfreeze^V :

$$\text{unfreeze}^V(\text{freeze}^V(T_V)) \triangleright T_V$$

Since $\text{freeze}^V(T_V)$ is a value, we can now pass “paused” computations to functions, and let these functions resume the computation if needed by using unfreeze^V . In a typed calculus, freeze^V would be the constructor of a type $\uparrow A_V$ called upshift, and unfreeze^V its destructor, as shown in Figure III.1.1.

Embedding $\lambda_V^{\rightarrow\uparrow}$ in λ_V^\rightarrow

Both freeze^V and unfreeze^V can actually be encoded in λ_V^\rightarrow so that there is a translation from $\lambda_V^{\rightarrow\uparrow}$ to λ_V^\rightarrow . The idea is that we can take $\text{freeze}^V(T_V) = \lambda x^V. T_V$ and $\text{unfreeze}^V(V_V) = V_V W_V$ where x^V is an arbitrary fresh variable, and W_V an arbitrary value. The reduction

$$\text{unfreeze}^V(\text{freeze}^V(T_V)) \triangleright T_V$$

then becomes

$$(\lambda_{-}^V. T_V) W_V \triangleright T_V$$

III. Pure polarized calculi

Figure III.1.1: The call-by-value λ -calculus with upshift $\lambda_v^{\rightarrow \uparrow}$

Figure III.1.1.a: Syntax

Values:
 $V_v, W_v ::= x^v$
 $\quad \mid \lambda x^v. T_v$
 $\quad \mid \text{freeze}^v(T_v)$
Terms:
 $T_v, U_v ::= \text{val}^v(V_v) \mid \text{let } x^v := T_v \text{ in } U_v$
 $\quad \mid T_v V_v$
 $\quad \mid \text{unfreeze}^v(T_v)$

Figure III.1.1.b: Top-level reduction

$$\begin{aligned} (\lambda x^v. T_v) V_v &\triangleright_{\text{let}} T_v[V_v/x^v] \\ \text{let } x^v := V_v \text{ in } T_v &\triangleright_{\rightarrow} T_v[V_v/x^v] \\ \text{unfreeze}^v(\text{freeze}^v(T_v)) &\triangleright_{\uparrow} T_v \\ \triangleright &\stackrel{\text{def}}{=} \triangleright_{\text{let}} \cup \triangleright_{\rightarrow} \cup \triangleright_{\uparrow} \end{aligned}$$

Figure III.1.1.c: Operational contexts

Operational contexts:
 $\mathcal{O}_v ::= \square \mid \text{let } x^v := \mathcal{O}_v \text{ in } T_v$
 $\quad \mid \mathcal{O}_v V_v$
 $\quad \mid \text{unfreeze}^v(T_v)$

Figure III.1.1.d: Typing rules

$$\dots \quad \frac{\Gamma \vdash T_v : A_v}{\Gamma \vdash \text{freeze}^v(T_v) : \uparrow A_v} \quad \frac{\Gamma \vdash T_v : \uparrow A_v}{\Gamma \vdash \text{unfreeze}^v(T_v) : A_v}$$

III. Pure polarized calculi

In call-by-value programming languages that have a **1** or **unit** type with a unique inhabitant $()^V$, it is common to take $\text{freeze}^V(T_V) = \lambda()^V.T_V$ and $\text{unfreeze}^V(V_V) = V_V()^V$ which works for the same reasons, but has two additional advantages: there are no arbitrary choices for the variable x^V and the value W_V , and the fact that x^V is not free in T_V is easier to see. In expressions of types, this means that we can encode $\uparrow A_V$ as $\uparrow A_V = 1 \rightarrow_V A_V$ (which is how $\uparrow A$ is defined in [CurFioMun16]). The definition operational reduction and contexts can be understood through this encoding: we reduce under $\text{unfreeze}^V(\square)$ because we reduce under $\square()$ and we do not reduce under $\text{freeze}^V(\square)$ because we do not reduce under $\lambda()^V.\square$.

Embedding λ_N^{\rightarrow} in $\lambda_V^{\rightarrow\uparrow}$

Figure III.1.2: Translations from λ_N^{\rightarrow} to $\lambda_V^{\rightarrow\uparrow}$

$$\begin{aligned} \dot{_}_V : \mathbf{T}_N &\rightarrow \mathbf{V}_V \\ \underline{x^N}_V &\stackrel{\text{def}}{=} x^V \\ \underline{\lambda x^N.T_N}_V &\stackrel{\text{def}}{=} \text{freeze}^V(\lambda x^V.\text{unfreeze}^V(\underline{T_N}_V)) \\ \underline{T_N U_N}_V &\stackrel{\text{def}}{=} \text{freeze}^V(\text{unfreeze}^V(\underline{T_N}_V) \underline{U_N}_V) \\ \underline{\text{let } x^N := T_N \text{ in } U_N}_V &\stackrel{\text{def}}{=} \text{freeze}^V(\text{let } x^V := \underline{T_N}_V \text{ in } \text{unfreeze}^V(\underline{U_N}_V)) \end{aligned}$$

The translation $\dot{_}_V$ described in Figure III.1.2 embeds λ_N^{\rightarrow} into $\lambda_V^{\rightarrow\uparrow}$ by freezing all computations, and only unfreezing them when they are on the left of an application. The evaluation of an expression T_N is simulated by the evaluation of $\text{unfreeze}^V(\underline{T_N}_V)$, e.g.

$$\begin{aligned} \text{unfreeze}^V(\underline{(\lambda x^N.T_N) U_N}_V) &= \text{unfreeze}^V(\text{freeze}^V(\text{unfreeze}^V(\text{freeze}^V(\lambda x^V.\text{unfreeze}^V(\underline{T_N}_V)))) \underline{U_N}_V) \\ &\triangleright_{\uparrow} \text{unfreeze}^V(\text{freeze}^V(\lambda x^V.\text{unfreeze}^V(\underline{T_N}_V))) \underline{U_N}_V \\ &\triangleright_{\uparrow} (\lambda x^V.\text{unfreeze}^V(\underline{T_N}_V)) \underline{U_N}_V \\ &\triangleright_{\rightarrow} \text{unfreeze}^V(\underline{T_N}_V[\underline{U_N}_V/x^V]) \\ &= \text{unfreeze}^V(\underline{T_N}_V[\underline{U_N}_V/x^N]) \end{aligned}$$

and

$$\begin{aligned} \text{unfreeze}^V(\underline{\text{let } x^N := T_N \text{ in } U_N}_V) &= \text{unfreeze}^V(\text{freeze}^V(\text{let } x^V := \underline{T_N}_V \text{ in } \text{unfreeze}^V(\underline{U_N}_V))) \\ &\triangleright_{\uparrow} \text{let } x^V := \underline{T_N}_V \text{ in } \text{unfreeze}^V(\underline{U_N}_V) \\ &\triangleright_{\text{let}} \text{unfreeze}^V(\underline{U_N}_V[\underline{T_N}_V/x^V]) \\ &= \text{unfreeze}^V(\underline{U_N}_V[\underline{T_N}_V/x^N]) \end{aligned}$$

[CurFioMun16] “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”, Curien, Fiore, and Munch-Maccagnoni, 2016

III. Pure polarized calculi

where the equalities

$$\frac{T_N[U_N/x^N]}{\vdash_v} = \frac{T_N[U_N/x^V]}{\vdash_v}$$

come from \vdash_v mapping variables x^N to variables x^V .

III.1.2. Embedding call-by-value in call-by-name

The $\lambda_N^{\rightarrow\Downarrow}$ -calculus

The extension of λ_N^{\rightarrow} , called $\lambda_N^{\rightarrow\Downarrow}$, is described in Figure III.1.3. The idea is that in λ_N^{\rightarrow} there is no way of distinguishing a value $\lambda x^N.T_N$ from an arbitrary expression U_N because

$$U_N \approx_\eta \lambda x^N.U_N.x^N$$

and two η -convertible expressions can not be distinguished. We therefore add a way to “mark” an expression T_N and a way of forcing evaluating to a “marked” expression:

$$\text{box}^N(T_N) \quad \text{and} \quad \text{match } T_N \text{ with } [\text{box}^N(x^N).U_N]$$

In a typed calculus, box^N would be the constructor of a type $\Downarrow A$ called downshift, and $\text{match } T_N \text{ with } [\text{box}^N(x^N).U_N]$ its associated pattern-match, as shown in Figure III.1.3.

Note that the pattern-match can be used to define a destructor

$$\text{unbox}^N(T_N) \stackrel{\text{ntn}}{=} \text{match } T_N \text{ with } [\text{box}^N(x^N).x^N]$$

with the expected induced reduction

$$\text{unbox}^N(\text{box}^N(T_N)) \triangleright_\Downarrow T_N$$

The destructor, however, can not be used to define the pattern-match. Indeed, while one could try to define the pattern-match

$$\text{match } T_N \text{ with } [\text{box}^N(x^N).x^N] \quad \text{as} \quad \text{let } x^N := \text{unbox}^N(T_N) \text{ in } U_N,$$

this would not work because the let-expression is call-by-name and will hence immediately reduce to $U_N[\text{unbox}^N(T_N)/x^N]$ while the match would first reduce T_N until it reaches an expression of the shape $\text{box}^N(T'_N)$. Note that in a call-by-value calculus, the pattern-match could be expressed using the destructor because $\text{let } x^V := \text{unbox}^V(T_V) \text{ in } U_V$ would also start by reducing T_V as expected.

Embedding $\lambda_N^{\rightarrow\Downarrow}$ in $\lambda_N^{\rightarrow\otimes}$ This box^N operator is not really common in programming languages but some other constructors are. For example, many calculi have pairs and the corresponding pattern match

$$(V_N \otimes W_N) \quad \text{and} \quad \text{match } T_N \text{ with } [(x^N \otimes y^N).U_N]$$

with the reduction

$$\text{match}(V_N \otimes W_N) \text{ with } [(x^N \otimes y^N).U_N] \triangleright_\otimes U_N[V_N/x^N, W_N/y^N]$$

Given those, constructor $\text{box}^N(T_N)$ can then be encoded as $(T_N \otimes V_N)$ where V_N is an arbitrary expression, and the match $\text{match } T_N \text{ with } [\text{box}^N(x^N).U_N]$ by $\text{match } T_N \text{ with } [(x^N \otimes y^N).U_N]$ with y^N fresh. Just like when encoding $\text{freeze}^V(T_V)$ as $\lambda()^V.T_V$ instead of $\lambda x^V.T_V$, the intended behavior becomes more apparent by replacing unused variables and values by $()^N$, so that $\text{box}^N(T_N)$ becomes $(T_N \otimes ()^N)$ and $\text{match } T_N \text{ with } [\text{box}^N(x^N).U_N]$ becomes $\text{match } T_N \text{ with } [(x^N \otimes ()^N).U_N]$.

III. Pure polarized calculi

Figure III.1.3: The $\lambda_N^{\rightarrow\Downarrow}$ calculus

Figure III.1.3.a: Syntax

Terms / values:
 $T_N, U_N, V_N, W_N ::= x^N \mid \text{let } x^N := T_N \text{ in } U_N$
 $\mid \lambda x^N. T_N \mid T_N U_N$
 $\mid \text{box}^N(V_N) \mid \text{match } T_N \text{ with } [\text{box}^N(x^N). U_N]$

Figure III.1.3.b: Top-level reduction

$\text{let } x^N := T_N \text{ in } U_N \triangleright_{\text{let}} U_N[T_N/x^N]$
 $(\lambda x^N. T_N) U_N \triangleright_{\rightarrow} T_N[U_N/x^N]$
 $\text{match } \text{box}^N(V_N) \text{ with } [\text{box}^N(x^N). U_N] \triangleright_{\Downarrow} U_N[V_N/x^N]$
 $\triangleright \stackrel{\text{def}}{=} \triangleright_{\text{let}} \cup \triangleright_{\rightarrow} \cup \triangleright_{\Downarrow}$

Figure III.1.3.c: Operational contexts

Operational contexts:
 $\mathbb{O}_N ::= \square$
 $\mid \mathbb{O}_N T_N$
 $\mid \text{match } \mathbb{O}_N \text{ with } [\text{box}^N(x^N). U_N]$

Figure III.1.3.d: Notations and induced reductions

$\text{unbox}^N(T_+) \stackrel{\text{ntn}}{=} \text{match } T_N \text{ with } [\text{box}^N(x^N). x^N]$ $\text{unbox}^N(\text{box}^N(V_N)) \triangleright_{\Downarrow} V_N$

Figure III.1.3.e: Typing rules

...

$\frac{\Gamma \vdash T_N : A_N}{\Gamma \vdash \text{box}^N(T_N) : \Downarrow A_N}$	$\frac{\Gamma \vdash T_N : \Downarrow A_N \quad \Gamma, x^N : A_N \vdash U_N : B_N}{\Gamma \vdash \text{match } T_N \text{ with } [\text{box}^N(x^N). U_N] : B_N}$
--	--

III. Pure polarized calculi

In a typed calculus, this would correspond to encoding $\Downarrow A_N$ as $\Downarrow A_N = A_N \otimes 1$ (which is how $\Downarrow A$ is defined in [CurFioMun16]).

Figure III.1.4: Translation from λ_V^{\rightarrow} to $\lambda_N^{\rightarrow\Downarrow}$

$$\begin{aligned}
v_N \cdot & : V_V \rightarrow T_N \\
v_N x^V & \stackrel{\text{def}}{=} x^N \\
v_N \lambda x^V. T_V & \stackrel{\text{def}}{=} \lambda x^N. v_N T_V \\
t_N \cdot & : T_V \rightarrow T_N \\
t_N \text{val}^V(V_V) & \stackrel{\text{def}}{=} \text{box}^N(v_N V_V) \\
t_N T_V V_V & \stackrel{\text{def}}{=} \text{unbox}^V(t_N T_V)_{v_N} V_V \\
t_N \text{let } x^V := T_V \text{ in } U_V & \stackrel{\text{def}}{=} \text{match}_{t_N} T_V \text{ with } [\text{box}^N(x^N). t_N T_N]
\end{aligned}$$

Embedding λ_V^{\rightarrow} in $\lambda_N^{\rightarrow\Downarrow}$ The translation from λ_V^{\rightarrow} to $\lambda_N^{\rightarrow\Downarrow}$ is described in Figure III.1.4. The idea is to translate values as expected with the $v_N \cdot$ part of the translation, and then use box^N to mark values, i.e. we translate val^V by box^N . We then extract the actual value when applying it or substituting it for a variable. The evaluation of T_V is simulated by the evaluation of $t_N T_V$, e.g.

$$\begin{aligned}
& t_N (\lambda x^V. T_V) V_V && t_N T_V [V_V / x^V] \\
& \parallel && \parallel \\
& \text{unbox}^V(\text{box}^V(\lambda x^N. t_N T_V))_{v_N} V_V \triangleright_{\Downarrow} (\lambda x^N. t_N T_V)_{v_N} V_V \triangleright_{\rightarrow} t_N T_V [v_N V_V / x^N]
\end{aligned}$$

and

$$\begin{aligned}
& t_N \text{let } x^V := V_V \text{ in } U_V && t_N U_V [V_V / x^V] \\
& \parallel && \parallel \\
& \text{match } \text{box}^N(v_N V_V) \text{ with } [\text{box}^N(x^N). t_N U_N] \triangleright_{\Downarrow} t_N U_N [v_N V_V / x^N]
\end{aligned}$$

One way to think of this translation in the well-typed fragment is that box^N and its pattern-match provide a runnable monad [ErwRen04] as explained in [Mun13; Mun14]. A computation of type A is represented as an element of $MA = \Downarrow A$, and the monad M has an extra operation $\text{run} : MA \rightarrow A$ that runs the computation, in addition to the usual ones: $\text{return} : A \rightarrow MA$ and $\text{bind} : MA \rightarrow (A \rightarrow MB) \rightarrow MB$. Here, return is box^N , bind (T_N, U_N) is $\text{match } T_N \text{ with } [\text{box}^N(x^N). U_N x^N]$ and run is unbox^N .

[CurFioMun16] “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”, Curien, Fiore, and Munch-Maccagnoni, 2016

[ErwRen04] “Monadification of Functional Programs”, Erwig and Ren, 2004

[Mun13] “Syntax and Models of a non-Associative Composition of Programs and Proofs”, Munch-Maccagnoni, 2013

[Mun14] “Models of a Non-Associative Composition”, Munch-Maccagnoni, 2014

III. Pure polarized calculi

III.2. A pure polarized λ -calculus: $\lambda_p^{\rightarrow\uparrow\downarrow}$



III. Pure polarized calculi

III.3. A pure polarized λ -calculus with focus: $\lambda_p^{\rightarrow\uparrow\downarrow}$



III. Pure polarized calculi

III.4. A pure polarized intuitionistic L-calculus: $\text{Li}_p^{\rightarrow\uparrow\downarrow}$



III. Pure polarized calculi

III.5. A pure polarized classical L-calculus: $L_p^{\rightarrow \uparrow \downarrow}$



IV. Polarized calculi with pairs and sums

IV.1. A polarized λ -calculus with pairs and sums: $\lambda_{\mathbf{p}}^{\rightarrow \& \uparrow \otimes \oplus \downarrow}$



IV.2. CBPV as a subcalculus of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$

Call-by-push-value (CBPV) [Lev01; Lev04; Lev06] is a well-known calculus that subsumes both call-by-name and call-by-value (including in the presence of side effects). It does so by decomposing Moggi’s computation monad [Mog89] as an adjunction. Typed models of LJ_p^η (i.e. of $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$) have been shown to generalize that of CBPV in [CurFioMun16]. In this section, we explain how $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ can be thought of as being CBPV “completed” by adding positive expressions, and in Section IV.6, we will explain how the CBPV abstract machine relates to $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$.

IV.2.1. CBPV

Syntax

We recall the syntax of Call-by-Push-Value (CBPV) in Figure IV.2.1a¹, with a few minor differences: we only have binary sum and negative pairs (and not those of arbitrary finite arity), we write $(V_{pv}, W_{pv})^{pv}$ for a pair instead of $\langle V, W \rangle$, and we add pv subscripts and superscripts. In CBPV, $\text{return}(V_{pv})$ is sometimes called $\text{produce}(V_{pv})$, and application $T_{pv} V_{pv}$ and (resp. projection $T_{pv} i$) are sometimes written in the reverse order $V_{pv} \text{ } T_{pv}$ (resp. $i \text{ } T_{pv}$).

Operational semantics

We recall the big-step operational semantics of CBPV Figure IV.2.1d², where $T_{pv} \Downarrow R_{pv}$ stands for “the computation T_{pv} terminates and its result is R_{pv} ”. Results form a subset of the set of computations, and their grammar is described in Figure IV.2.1c.

Complex values

Figure IV.2.1b³ extends CBPV with complex values such as $\text{pm } x^{pv} \text{ as } [(y^{pv}, z^{pv})^{pv}]. y^{pv}$. These are useful when looking at the semantics of CBPV, but not suitable for operational semantics because “they detract from the rigid sequential nature of the language, because they can be evaluated at any time” [Lev06]. Adding complex values has no effect on what computations can be expressed (see Proposition 14 of [Lev06]), because CBPV with complex values can be translated to CBPV without complex values (see Figure 13 of [Lev06]).

¹This figure corresponds to figure 3.1 of [Lev01], figure 2.1 of [Lev04], figure 2 of [Lev06].

²This figure corresponds to Figure 4 of [Lev06].

³This figure corresponds to Figure 12 of [Lev06].

[Lev01] “Call-by-push-value”, Levy, 2001

[Lev04] *Call-By-Push-Value: A Functional/Imperative Synthesis*, Levy, 2004

[Lev06] “Call-by-push-value: Decomposing call-by-value and call-by-name”, Levy, 2006

[Mog89] “Computational Lambda-Calculus and Monads”, Moggi, 1989

[CurFioMun16] “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”, Curien, Fiore, and Munch-Maccagnoni, 2016

Figure IV.2.1: Call-by-Push-Value

Figure IV.2.1.a: Syntax

Values:
 $V_{pv} ::= x^{pv}$
 $| (V_{pv}, W_{pv})^{pv}$
 $| (1, V_{pv})^{pv} | (2, V_{pv})^{pv}$
 $| \text{thunk}(T_{pv})$

Expressions / computations:
 $T_{pv}, U_{pv} ::= V_{pv} | \text{let } x^{pv} \text{ be } V_{pv} . U_{pv}$
 $| \lambda x^{pv} . T_{pv} | T_{pv} V_{pv}$
 $| \lambda^{pv} [1. T_{pv} | 2. U_{pv}] | T_{pv} 1 | T_{pv} 2$
 $| \text{return}(V_{pv}) | T_{pv} \text{ to } x^{pv} . U_{pv}$
 $| \text{pm } V_{pv} \text{ as } [(x^{pv}, y^{pv})^{pv} . U_{pv}]$
 $| \text{pm } V_{pv} \text{ as } [(1, x_1^{pv})^{pv} . U_{pv}^1 | (2, x_2^{pv})^{pv} . U_{pv}^2]$
 $| \text{force}(V_{pv})$

Figure IV.2.1.b: Syntax with complex values

Complex values:
 $V_{cv}, W_{cv} ::= x^{pv} | \text{let } x^{pv} \text{ be } V_{cv} . W_{cv}$
 $| (V_{cv}, W_{cv})^{pv} | \text{pm } V_{cv} \text{ as } [(x^{pv}, y^{pv})^{pv} . W_{cv}]$
 $| (1, V_{cv})^{pv} | (2, V_{cv})^{pv} | \text{pm } V_{cv} \text{ as } [(1, x_1^{pv})^{pv} . W_{cv}^1 | (2, x_2^{pv})^{pv} . W_{cv}^2]$
 $| \text{thunk}(T_{cv})$

Expressions / computations (with complex values):
 $T_{cv}, U_{cv} ::= \left(\begin{array}{l} \text{Same production rules as } T_{pv} \\ \text{with all occurrences of } V_{pv} \text{ replaced by } V_{cv}. \\ \text{See Figure IV.2.2.} \end{array} \right)$

IV. Polarized calculi with pairs and sums

Figure IV.2.1.c: Syntax of results

Results:

$$R_{pv} ::= \text{return}(V_{pv}) \mid \lambda x^{pv}. T_{pv} \mid \lambda^{pv} [1. T_{pv}^1 \mid 2. T_{pv}^2]$$

Figure IV.2.1.d: Big-step operational semantics

$$\begin{array}{c}
\frac{T_{pv}[V_{pv}/x^{pv}] \Downarrow R_{pv}}{\text{let } x^{pv} \text{ be } V_{pv}. T_{pv} \Downarrow R_{pv}} \\
\\
\frac{}{\lambda x^{pv}. T_{pv} \Downarrow \lambda x^{pv}. T_{pv}} \quad \frac{T_{pv} \Downarrow \lambda x^{pv}. U_{pv} \quad U_{pv}[V_{pv}/x^{pv}] \Downarrow R_{pv}}{T_{pv} V_{pv} \Downarrow R_{pv}} \\
\\
\frac{}{\lambda^{pv} [1. T_{pv}^1 \mid 2. T_{pv}^2] \Downarrow \lambda^{pv} [1. T_{pv}^1 \mid 2. T_{pv}^2]} \quad \frac{T_{pv} \Downarrow \lambda^{pv} [1. U_{pv}^1 \mid 2. U_{pv}^2] \quad U_{pv}^i \Downarrow R_{pv}}{T_{pv} i \Downarrow R_{pv}} \\
\\
\frac{}{\text{return}(V_{pv}) \Downarrow \text{return}(V_{pv})} \quad \frac{T_{pv} \Downarrow \text{return}(V_{pv}) \quad U_{pv}[V_{pv}/x^{pv}] \Downarrow R_{pv}}{T_{pv} \text{ to } x^{pv}. U_{pv} \Downarrow R_{pv}} \\
\\
\frac{T_{pv} \Downarrow R_{pv}}{\text{force}(\text{thunk}(T_{pv})) \Downarrow R_{pv}} \\
\\
\frac{T_{pv}[V_{pv}/x^{pv}, W_{pv}/y^{pv}] \Downarrow R_{pv}}{\text{pm}(V_{pv}, W_{pv})^{pv} \text{ as } [(x^{pv}, y^{pv})^{pv}. T_{pv}] \Downarrow R_{pv}} \\
\\
\frac{T_{pv}^i[V_{pv}/x_i^{pv}] \Downarrow R_{pv}}{\text{pm}(i, V_{pv})^{pv} \text{ as } [(1, x_1^{pv})^{pv}. T_{pv}^1 \mid (2, x_2^{pv})^{pv}. T_{pv}^2] \Downarrow R_{pv}}
\end{array}$$

Figure IV.2.2: Syntax of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ (left) and CBPV (right)

Positive values:

$$\begin{aligned} V_+, W_+ ::= & x^+ \\ & | (V_+ \otimes W_+) \\ & | \iota_1(V_+) \mid \iota_2(V_+) \\ & | \text{box}(V_+) \end{aligned}$$

Negative values / expressions:

$$\begin{aligned} V_-, W_-, T_-, U_- ::= & x^- \mid \text{let } x^+ := T_+ \text{ in } U_- \mid \text{let } x^- := T_- \text{ in } U_- \\ & | \lambda x^+. T_- \mid T_- V_+ \\ & | (T_- \& U_-) \mid \pi_1(T_-) \mid \pi_2(T_-) \\ & | \text{freeze}(T_+) \\ & | \text{match } T_+ \text{ with } [(x^+ \otimes y^+). U_-] \\ & | \text{match } T_+ \text{ with } [\iota_1(x_1^+). U_-^1 \mid \iota_2(x_2^+). U_-^2] \\ & | \text{match } T_+ \text{ with } [\text{box}(x^-). U_-] \end{aligned}$$

Positive expressions:

$$\begin{aligned} T_+, U_+ ::= & V_+ \\ & | \text{let } x^+ := T_+ \text{ in } U_+ \mid \text{let } x^- := T_- \text{ in } U_+ \\ & | \text{unfreeze}(T_-) \\ & | \text{match } T_+ \text{ with } [(x^+ \otimes y^+). U_+] \\ & | \text{match } T_+ \text{ with } [\iota_1(x_1^+). U_+^1 \mid \iota_2(x_2^+). U_+^2] \\ & | \text{match } T_+ \text{ with } [\text{box}(x^+). U_+] \end{aligned}$$

Values:

$$\begin{aligned} V_{pv} ::= & x^{pv} \\ & | (V_{pv} \circ W_{pv})^{pv} \\ & | (1, V_{pv})^{pv} \mid (2, V_{pv})^{pv} \\ & | \text{thunk}(T_{pv}) \end{aligned}$$

Expressions / computations:

$$\begin{aligned} T_{pv}, U_{pv} ::= & V_{pv} \mid \text{let } x^{pv} \text{ be } V_{pv} \cdot U_{pv} \\ & | \lambda x^{pv}. T_{pv} \mid T_{pv} V_{pv} \\ & | \lambda^{pv} [1. T_{pv} \mid 2. U_{pv}] \mid T_{pv} 1 \mid T_{pv} 2 \\ & | \text{return}(V_{pv}) \mid T_{pv} \text{ to } x^{pv}. U_{pv} \\ & | \text{pm } V_{pv} \text{ as } [(x^{pv}, y^{pv})^{pv}. U_{pv}] \\ & | \text{pm } V_{pv} \text{ as } [(1, x_1^{pv})^{pv}. U_{pv}^1 \mid (2, x_2^{pv})^{pv}. U_{pv}^2] \\ & | \text{force}(V_{pv}) \end{aligned}$$

Complex values:

$$\begin{aligned} V_{cv}, W_{cv} ::= & x^{pv} \mid (V_{cv} \circ W_{cv})^{pv} \mid (1, V_{cv})^{pv} \mid (2, V_{cv})^{pv} \mid \text{thunk}(T_{pv}) \\ & | \text{let } x^{pv} \text{ be } V_{cv} \cdot W_{cv} \\ & | \\ & | \text{pm } V_{cv} \text{ as } [(x^{pv}, y^{pv})^{pv}. W_{cv}] \\ & | \text{pm } V_{cv} \text{ as } [(1, x_1^{pv})^{pv}. W_{cv} \mid (2, x_2^{pv})^{pv}. W_{cv}] \end{aligned}$$

IV.2.2. Embedding CBPV into $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$

Embedding values and computations

The syntaxes of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ and CBPV are shown side by side in Figure IV.2.2, with expressions and values that correspond to each other placed on the same line, and things that are present in one calculus but not the other highlighted. Values V_{pv} of CBPV correspond to positive values V_+ of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$, and expressions T_{pv} of CBPV correspond to negative expressions T_- of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$. For shifts (see Figure IV.2.3), $\text{thunk}(T_{pv})$ corresponds to $\text{box}(T_-)$, $\text{force}(V_{pv})$ to $\text{unbox}(V_+)$, and $\text{return}(V_{pv})$ to $\text{freeze}(\text{val}(V_+))$ (i.e. the restriction of the general $\text{freeze}(T_+)$ to values). The “inverse” T_{pv} to $x^{pv} \cdot U_{pv}$ of $\text{return}(V_{pv})$ corresponds to $\text{let } x^+ := \text{unfreeze}(T_-) \text{ in } U_-$. The values types A_{pv} and computation type B_{pv} of CBPV (which are not described here) correspond to positive types A_+ and negative types B_- respectively, with $F^{pv}(A_{pv})$ corresponding to $\uparrow A_+$ and $U^{pv}(B_{pv})$ to $\Downarrow B_-$ ⁴. More precisely, the translation

$$\cdot_p : \text{CBPV} \rightarrow \lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$$

described in Figure IV.2.4 is an embedding:

Fact IV.2.1

The translation $\cdot_p : \text{CBPV} \rightarrow \lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ is injective.

Proof

By induction on the syntax.

Fact IV.2.2

The translation \cdot_p is substitutive: for any computation T_{pv} (resp. value V_{pv}), variable x^{pv} , and value W_{pv} , we have

$$\frac{T_{pv}[W_{pv}/x^{pv}]}{\cdot_p} = \frac{T_{pv}[W_{pv}/x^+]}{\cdot_p} \quad (\text{resp. } \frac{V_{pv}[W_{pv}/x^{pv}]}{\cdot_p} = \frac{V_{pv}[W_{pv}/x^+]}{\cdot_p})$$

Proof

By induction on the syntax of T_{pv} (resp. V_{pv}).

Differences between CBPV and $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$

With these correspondances in mind, there are two main differences between CBPV and $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$:

⁴For this correspondance, one can remember that U^{pv} unfortunately *does not* corresponds to the \Downarrow shift \uparrow , or notice that both \supset and \Rightarrow are common symbols for implication, and that applying the same rotation to both of them yields U^{pv} and \Downarrow .

IV. Polarized calculi with pairs and sums

Figure IV.2.3: Shifts in $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$ (left) and CBPV (right)

$$\begin{array}{c} \mathbf{V}_+ \xleftarrow{\text{box}} \mathbf{V}_- = \mathbf{T}_- \\ \text{in} \\ \mathbf{T}_+ \xleftarrow{\text{freeze}} \end{array}$$

$$\begin{array}{c} \mathbf{V}_{pv} \xleftarrow{\text{thunk}} \mathbf{T}_{pv} \\ \text{return} \end{array}$$

Figure IV.2.4: Embedding $\dot{_}_p$ of CBPV into $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$

Values:

$$\begin{aligned} \dot{_}_p : \mathbf{V}_{pv} &\rightarrow \mathbf{V}_+ \\ \frac{x^{pv}}{_}_p &\stackrel{\text{def}}{=} x^+ \\ \frac{(V_{pv}, W_{pv})^{pv}}{_}_p &\stackrel{\text{def}}{=} \left(\frac{V_{pv}}{_}_p \otimes \frac{W_{pv}}{_}_p \right) \\ \frac{(i, V_{pv})^{pv}}{_}_p &\stackrel{\text{def}}{=} \iota_i \left(\frac{V_{pv}}{_}_p \right) \\ \frac{\text{thunk}(T_{pv})}{_}_p &\stackrel{\text{def}}{=} \text{box} \left(\frac{T_{pv}}{_}_p \right) \end{aligned}$$

Expressions:

$$\begin{aligned} \dot{_}_p : \mathbf{T}_{pv} &\rightarrow \mathbf{T}_- \\ \frac{\text{pm } V_{pv} \text{ as } [(x^{pv}, y^{pv})^{pv}. T_{pv}]}{_}_p &\stackrel{\text{def}}{=} \text{match } \frac{V_{pv}}{_}_p \text{ with } [(x^+ \otimes y^+). \frac{T_{pv}}{_}_p] \\ \frac{\text{pm } V_{pv} \text{ as } [(1, x_1^{pv})^{pv}. T_{pv}^1 \mid (2, x_2^{pv})^{pv}. T_{pv}^2]}{_}_p &\stackrel{\text{def}}{=} \text{match } \frac{V_{pv}}{_}_p \text{ with } [\iota_1(x_1^+). \frac{V_{pv}}{_}_p \mid \iota_2(x_2^+). \frac{T_{pv}^2}{_}_p] \\ \frac{\text{force}(V_{pv})}{_}_p &\stackrel{\text{def}}{=} \text{unbox} \left(\frac{V_{pv}}{_}_p \right) \\ \frac{\text{let } x^{pv} \text{ be } V_{pv}. T_{pv}}{_}_p &\stackrel{\text{def}}{=} \text{let } x^+ := \frac{V_{pv}}{_}_p \text{ in } \frac{T_{pv}}{_}_p \\ \frac{\lambda x^{pv}. T_{pv}}{_}_p &\stackrel{\text{def}}{=} \lambda x^+. \frac{T_{pv}}{_}_p \\ \frac{\frac{T_{pv} V_{pv}}{_}_p}{_}_p &\stackrel{\text{def}}{=} \frac{\frac{T_{pv}}{_}_p \otimes \frac{V_{pv}}{_}_p}{_}_p \\ \frac{\lambda^{pv} [1. T_{pv}^1 \mid 2. T_{pv}^2]}{_}_p &\stackrel{\text{def}}{=} \left(\frac{T_{pv}^1}{_}_p \& \frac{T_{pv}^2}{_}_p \right) \\ \frac{T_{pv} i}{_}_p &\stackrel{\text{def}}{=} \pi_i \left(\frac{T_{pv}}{_}_p \right) \\ \frac{\text{return}(V_{pv})}{_}_p &\stackrel{\text{def}}{=} \text{freeze} \left(\frac{V_{pv}}{_}_p \right) \\ \frac{T_{pv} \text{ to } x^{pv}. U_{pv}}{_}_p &\stackrel{\text{def}}{=} \text{let } x^+ := \text{unfreeze} \left(\frac{T_{pv}}{_}_p \right) \text{ in } \frac{U_{pv}}{_}_p \end{aligned}$$

IV. Polarized calculi with pairs and sums

- There are no negative variables x^- in CBPV, and hence no $\text{let } x^- := V_- \text{ in } U_-$. The only other use of negative variables in $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$, namely $\text{match } T_+ \text{ with } [\text{box}(x^-). U_-]$, is restricted to the cases where T_+ is a value $T_+ = V_+$ and $U_- = x^-$, i.e. to $\text{unbox}(V_+)$, and is denoted by $\text{force}(V_{pv})$.
- There are no non-value positive expressions T_+ in CBPV (without complex values), which corresponds to replacing T_+ by V_+ everywhere in the syntax of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$. Since $\text{unfreeze}(T_-)$ is a positive expression, it is no longer expressible, and is therefore replaced by $T_{pv} \text{ to } x^{pv}. U_{pv}$ which corresponds to its composition with a let-expression $\text{let } x^+ := \text{unfreeze}(T_+) \text{ in } U_-$.

Complex values and positive expressions

Complex values V_{cv} are very similar to positive expressions T_+ , but neither of the set is contained in the other:

- $\text{unfreeze}(T_-)$ corresponds to no complex value; and
- $((\lambda x^{pv}. \text{return}(x^{pv}))V_{pv}, W_{pv})^{pv}$ is a complex value, while $((\lambda x^+. \text{freeze}(x^+))V_+ \otimes W_+)$ is not a positive term (because $(\lambda x^+. \text{freeze}(x^+))V_+$ is not a value).

Complex values can nevertheless be represented by positive terms via let-expansions, e.g.

$((\lambda x^{pv}. \text{return}(x^{pv}))V_{pv}, W_{pv})^{pv}$ corresponds to $\text{let } y^+ := (\lambda x^+. \text{freeze}(x^+))V_+ \text{ in } (y^+ \otimes W_+)$

More generally, if V_{cv} corresponds to T_+ , and W_{cv} to U_+ , then the complex value

$(V_{cv}, W_{cv})^{pv}$ corresponds to $\text{let } x^+ := T_+ \text{ in let } y^+ := U_+ \text{ in } (x^+ \otimes y^+)$

Expanding CBPV with positive terms (i.e. to $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$) has the same advantages as extending it with complex values (i.e. it makes it better suited for semantic endeavors), but avoids the complications of the operational semantics induced by complex values: the choice of when to evaluate complex values is pushed to the “user” through the need for let-expression to express some complex values. Of course, in an actual programming language, we would want to be able to write $(T_+ \otimes U_+)$, but this could be a notation for $\text{let } x^+ := T_+ \text{ in let } y^+ := U_+ \text{ in } (x^+ \otimes y^+)$, and can therefore be ignored for theoretical purposes.

Preservation of operational semantics

In $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$, we have a small-step operational semantics \triangleright , which induces a big-step operational semantics given by

$$T_\varepsilon \Downarrow T'_\varepsilon \stackrel{\text{def}}{=} T_\varepsilon \triangleright^* T'_\varepsilon \triangleright$$

Through the translation $\underline{\cdot}_p$, the big-step operational semantics of CBPV corresponds exactly to the one of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$:

Proposition IV.2.3

For any closed expression $T_{pv}, T_{pv} \Downarrow R_{pv}$ if and only if $\frac{T_{pv}}{\longrightarrow_p} \Downarrow \frac{R_{pv}}{\longrightarrow_p}$.

IV. Polarized calculi with pairs and sums

Proof

- \Rightarrow We have $\frac{T_{pv}}{\longrightarrow_P} \triangleright^* \frac{R_{pv}}{\longrightarrow_P}$ by induction on the derivation of $T_{pv} \Downarrow R_{pv}$ and Fact IV.2.2, and $\frac{R_{pv}}{\longrightarrow_P} \not\triangleright$ by case analysis on the syntax of R_{pv} .
- \Leftarrow By induction on the length of the reduction $\frac{T_{pv}}{\longrightarrow_P} \triangleright^* \frac{R_{pv}}{\longrightarrow_P}$.

IV.3. A polarized λ -calculus with focus: $\lambda_{\perp}^{\rightarrow \& \uparrow \otimes \oplus \downarrow}$



IV.4. A polarized intuitionistic L calculus: $\text{Li}_p^{\rightarrow \& \uparrow \otimes \oplus \downarrow}$



IV.5. A polarized classical L calculus: $L_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$



IV.6. The CBPV abstract machine as a subcalculus of

$\lambda_{\text{p}}^{\rightarrow \& \uparrow \otimes \oplus \downarrow}$



V. Polarized calculi with arbitrary constructors

V.1. A (classical) polarized L-calculus: $L_p^{\vec{\tau}}$	104
V.2. Intuitionistic and minimalistic polarized L-calculi: $Li_p^{\vec{\tau}}$ and $Lm_p^{\vec{\tau}}$	126
V.3. A polarized λ -calculus with focus equivalent to $Lm_p^{\vec{\tau}}$: $\lambda_p^{\vec{\tau}}$	147
V.4. Equivalence between $\lambda_p^{\vec{\tau}}$ and $Lm_p^{\vec{\tau}}$	148
V.5. A polarized λ -calculus: $\lambda_p^{\vec{\tau}}$	149

V.1. A (classical) polarized L-calculus: $L_p^{\vec{\tau}}$

V.1.1. Syntax

Type formers

Everything starts with a (finite) set of positive type formers $\tau_+^1, \dots, \tau_+^n$ and negative type formers $\tau_-^1, \dots, \tau_-^m$ that generate positive types A_+ and negative types A_- as described in Figure V.1.1a. With the usual type formers, this yields Figure V.1.1e. For binary type formers (e.g. \rightarrow), we often use the infix notation (e.g. we write $A_+ \rightarrow B_-$ for $\rightarrow(A_+, B_-)$). Even though the notation $\tau_\epsilon^j(\vec{A})$ may suggested it, the type formers τ_ϵ^j do not take arbitrary sequence \vec{A} of arguments: the arity of each τ_ϵ^j is fixed (e.g. $\rightarrow(A_-)$ would be invalid), and the polarity of each argument is also fixed (e.g. $\rightarrow(A_+, B_+)$ would also be invalid). A more precise notation would be

$$\tau_\epsilon^j(A_{\text{pol}(\tau_\epsilon^j, 1)}^1, \dots, A_{\text{pol}(\tau_\epsilon^j, \text{ar}(\tau_\epsilon^j))}^{\text{ar}(\tau_\epsilon^j)})$$

where $\text{ar}(\tau_\epsilon^j)$ is the arity of τ_ϵ^j , and $\text{pol}(\tau_\epsilon^j, k)$ is the polarity of the k^{th} argument of τ_ϵ^j . In other words, when we write $\tau_\epsilon^j(\vec{A})$, the length and shape of \vec{A} depends on τ_ϵ^j , but we do not make this dependence explicit in the notations.

Value and stack constructors

We denote by a and call argument any value v or stack s , and write \vec{a} for an arbitrary list a_1, \dots, a_q of arguments. We denote by χ^1 and call variable any value variable x^ϵ or stack variable α^ϵ , and write $\vec{\chi}$ for an arbitrary list χ_1, \dots, χ_q of variables.

As depicted in Figure V.1.2, each positive type former τ_+^j (resp. negative type former τ_-^j) has $l_j^+ \in \mathbb{N}$ (positive) value constructors $\mathfrak{b}_1^{\tau_+^j}, \dots, \mathfrak{b}_{l_j^+}^{\tau_+^j}$ (resp. $l_j^- \in \mathbb{N}$ (negative) stack constructors $\mathfrak{s}_1^{\tau_-^j}, \dots, \mathfrak{s}_{l_j^-}^{\tau_-^j}$), which can be applied to suitable arguments to form positive values $\mathfrak{b}_k^{\tau_+^j}(\vec{a})$ (resp. negative stacks $\mathfrak{s}_k^{\tau_-^j}(\vec{a})$)², and a positive stack (resp. negative value)

$$\tilde{\mu} \left[\begin{array}{c} \mathfrak{b}_1^{\tau_+^j}(\vec{\chi}_1).c_1 \\ \vdots \\ \mathfrak{b}_{l_j^+}^{\tau_+^j}(\vec{\chi}_{l_j^+}).c_{l_j^+} \end{array} \right] \quad \left(\text{resp. } \mu \left\langle \begin{array}{c} \mathfrak{s}_1^{\tau_-^j}(\vec{\chi}_1).c_1 \\ \vdots \\ \mathfrak{s}_{l_j^-}^{\tau_-^j}(\vec{\chi}_{l_j^-}).c_{l_j^-} \end{array} \right\rangle \right)$$

that matches over all values (resp. stacks) formed using these constructors. This stack (resp. value) is often denoted by

$$\tilde{\mu} \left[\mathfrak{b}_1^{\tau_+^j}(\vec{\chi}_1).c_1 \mid \dots \mid \mathfrak{b}_{l_j^+}^{\tau_+^j}(\vec{\chi}_{l_j^+}).c_{l_j^+} \right] \quad \left(\text{resp. } \mu \left\langle \mathfrak{s}_1^{\tau_-^j}(\vec{\chi}_1).c_1 \mid \dots \mid \mathfrak{s}_{l_j^-}^{\tau_-^j}(\vec{\chi}_{l_j^-}).c_{l_j^-} \right\rangle \right)$$

When $l_j^+ = 1$ (resp. $l_j^- = 1$), we sometimes write these without the $[\cdot]$ (resp. $\langle \cdot \rangle$), e.g.

¹Mnemonic: the symbol for variables χ looks like the symbol for value variables x and is from the Greek alphabet just like the symbol for stack variables α .

²Note that even though we write $\mathfrak{b}_k^{\tau_+^j}(\vec{a})$ (resp. $\mathfrak{s}_k^{\tau_-^j}(\vec{a})$), the length and shape of \vec{a} depend on τ_+^j and $\mathfrak{b}_k^{\tau_+^j}$ (resp. τ_-^j and $\mathfrak{s}_k^{\tau_-^j}$), just like we wrote $\tau_\epsilon^j(\vec{A})$ even though the length and shape of \vec{A} could depend on τ_ϵ^j .

Figure V.1.1: Types generated by a set of type formers $\vec{\tau}$

Figure V.1.1.a: Types generated by $\vec{\tau} = \tau_+^1 \dots \tau_+^n \tau_-^1 \dots \tau_-^m$

Positive types:	Negative types:
$A_+, B_+ ::= \tau_+^1(\vec{A})$	$A_-, B_- ::= \tau_-^1(\vec{A})$
$ \vdots$	$ \vdots$
$ \tau_+^n(\vec{A})$	$ \tau_-^m(\vec{A})$

Figure V.1.1.b: Types generated by $\vec{\tau} = \rightarrow_-$

Positive types:	Negative types:
$A_+, B_+ ::= (\text{none})$	$A_-, B_- ::= A_- \rightarrow_- B_-$

Figure V.1.1.c: Types generated by $\vec{\tau} = \rightarrow_+ \Downarrow$

Positive types:	Negative types:
$A_+, B_+ ::= \Downarrow A_-$	$A_-, B_- ::= A_+ \rightarrow_+ B_+$

Figure V.1.1.d: Types generated by $\vec{\tau} = \rightarrow \Downarrow \Uparrow$

Positive types:	Negative types:
$A_+, B_+ ::=$ $ \Downarrow A_-$	$A_-, B_- ::= A_+ \rightarrow B_-$ $ \Uparrow A_+$

Figure V.1.1.e: Types generated by $\vec{\tau} = \rightarrow \Downarrow \Uparrow \neg_+ \neg_- \otimes \wp \oplus \& 1 \perp 0 \top$

Positive types:	Negative types:
$A_+, B_+ ::=$ $ \Downarrow A_-$ $ \neg_+(A_-)$ $ A_+ \otimes B_+$ $ A_+ \oplus B_+$ $ 1$ $ 0$	$A_-, B_- ::= A_+ \rightarrow B_-$ $ \Uparrow A_+$ $ \neg_-(A_+)$ $ A_- \wp B_-$ $ A_- \& B_-$ $ \perp$ $ \top$

Figure V.1.2: Examples of value and stack constructors

Figure V.1.2.a: Examples of value constructors and value pattern-matchings

Positive type former	Value constructors	Value pattern match
\Downarrow	$\mathfrak{v}_1^\Downarrow(v_-) = \{v_-\}$	$\tilde{\mu}\{x^-\}.c$
\neg_+	$\mathfrak{v}_1^{\neg_+}(s_-) = \neg_+(s_-)$	$\tilde{\mu}\neg_+(\alpha^-).\alpha c$
\otimes	$\mathfrak{v}_1^\otimes(v_+, w_+) = (v_+ \otimes w_+)$	$\tilde{\mu}(x^+ \otimes y^+).c$
\oplus	$\mathfrak{v}_1^\oplus(v_+) = \iota_1(v_+)$ $\mathfrak{v}_2^\oplus(v_+) = \iota_2(v_+)$	$\tilde{\mu}\left[\iota_1(x_1^+).c_1\right]$ $\tilde{\mu}\left[\iota_2(x_2^+).c_2\right]$
1	$\mathfrak{v}_1^1() = ()$	$\tilde{\mu}().c$
0	(none)	$\tilde{\mu}[]$

Figure V.1.2.b: Examples of stack constructors and stack pattern-matchings

Negative type former	Stack constructors	Stack pattern match
\rightarrow	$\mathfrak{s}_1^{\rightarrow}(v_+, s_-) = v_+ \cdot s_-$	$\mu(x^+ \cdot \alpha^-).c$
\rightarrow_-	$\mathfrak{s}_1^{\rightarrow_-}(v_-, s_-) = v_- \cdot s_-$	$\mu(x^- \cdot \alpha^-).c$
\rightarrow_+	$\mathfrak{s}_1^{\rightarrow_+}(v_+, s_+) = v_+ \cdot s_+$	$\mu(x^+ \cdot \alpha^+).c$
\Uparrow	$\mathfrak{s}_1^{\Uparrow}(s_+) = \{s_+\}$	$\mu\{x^+\}.c$
\neg_-	$\mathfrak{s}_1^{\neg_-}(v_+) = \neg_-(v_+)$	$\mu\neg_-(x^+).\alpha c$
\wp	$\mathfrak{s}_1^{\wp}(s_-^1, s_-^2) = (s_-^1 \wp s_-^2)$	$\mu(\alpha^- \wp \beta^-).c$
$\&$	$\mathfrak{s}_1^{\&}(s_-) = \pi_1 \cdot s_-$ $\mathfrak{s}_2^{\&}(s_-) = \pi_2 \cdot s_-$	$\mu\left\langle(\pi_1 \cdot \alpha_1^-).c_1\right\rangle$ $\mu\left\langle(\pi_2 \cdot \alpha_2^-).c_2\right\rangle$
\perp	$\mathfrak{s}_1^\perp() = \tilde{()}$	$\mu\tilde{()}.c$
\top	(none)	$\mu\langle\rangle$

writing

$$\tilde{\mu}\{x^+\}.c \text{ for } \tilde{\mu}[\{x^+\}.c] \quad (\text{resp. } \mu\{\alpha^-\}.c \text{ for } \mu\langle\{\alpha^-\}.c\rangle)$$

To simplify notations, we sometimes assume that constructors take value arguments before stack arguments:

Definition V.1.1

A constructor $\mathfrak{b}_k^{\tau_j^+}$ (resp. $\mathfrak{s}_k^{\tau_j^-}$) is said to be *vs-sorted* when its value arguments are on the left of its stack arguments, i.e. when

$$\mathfrak{b}_k^{\tau_j^+}(\vec{a}) = \mathfrak{b}_k^{\tau_j^+}(\vec{v}, \vec{s}) \quad (\text{resp. } \mathfrak{s}_k^{\tau_j^-}(\vec{a}) = \mathfrak{s}_k^{\tau_j^-}(\vec{v}, \vec{s}))$$

Replacing a constructor $\mathfrak{b}_k^{\tau_j^+}$ (resp. $\mathfrak{s}_k^{\tau_j^-}$) by another one that takes its arguments in another order changes nothing for our purposes, and we therefore assume that all constructors are vs-sorted when convenient.

Syntax

The syntax of $L_p^{\vec{\tau}}$ is given in Figure V.1.3a, and the result of instantiating it with $\vec{\tau} = \rightarrow \downarrow \uparrow \neg \neg_+ \otimes \wp \oplus \& 1 \perp 0 \top$ is given in Figure V.1.4a. The polarities ε on commands $\langle \cdot \rangle^\varepsilon$, and $+$ and $-$ on the coercions val^+ and stk^- are there to ensure that the induced grammar of fragments (see \triangleleft) is non-ambiguous, but are superfluous in the grammar of terms (i.e. removing them does not make the grammar of terms ambiguous). The coercions val^+ and stk^- are often left implicit³.

V.1.2. Reductions

Definitions

The operational reduction \triangleright (which is also the top-level β -reduction \triangleright in $L_p^{\vec{\tau}}$) is defined defined in Figure V.1.3c, and the top-level η -expansion \mathfrak{j} is defined in Figure V.1.3d. The result of instantiating these with $\vec{\tau} = \rightarrow \downarrow \uparrow \neg \neg_+ \otimes \wp \oplus \& 1 \perp 0 \top$ is described in Figure V.2.3b and Figure V.2.3c respectively. The strong reduction \rightarrow is defined as the contextual closure $\mathcal{K} \boxtimes$ of the operational reduction \triangleright , and the η -expansion \rightarrow as the contextual closure $\mathcal{K} \mathfrak{j}$ of the top-level η -expansion \mathfrak{j} . The reduction \rightarrow^0 is defined as the closure $(\mathcal{K} \setminus \{\square\}) \boxtimes$ of the operational reduction \triangleright under non-trivial contexts. Alternative definitions of these closures via inference rules can be found in \triangleleft .

³We only use these coercions when defining the η -expansions \mathfrak{j}_μ and $\mathfrak{j}_{\bar{\mu}}$ (see Remark V.1.2), and for everything else, we leave these coercions implicit.

Figure V.1.3: The L_p^τ calculus

Figure V.1.3.a: Syntax

Positive values:

$$v_+, w_+ ::= x^+$$

$$| \mathfrak{b}_1^{\tau_1^+}(\vec{a}) | \dots | \mathfrak{b}_{l_+^+}^{\tau_+^+}(\vec{a}) |$$

$$| \vdots | \quad | \cdot \vdots | \quad | \vdots |$$

$$| \mathfrak{b}_1^{\tau_1^n}(\vec{a}) | \dots | \mathfrak{b}_{l_n^n}^{\tau_n^n}(\vec{a}) |$$

Positive expressions:

$$t_+, u_+ ::= \text{val}^+(v_+) \mid \mu\alpha^+.c$$

Negative values / expressions:

$$v_-, w_-, t_-, u_- ::= x^- \mid \mu\alpha^-.c$$

$$| \mu \langle \mathfrak{s}_1^{\tau_1^+}(\vec{\chi}_1).c_1 | \dots | \mathfrak{s}_{l_+^+}^{\tau_+^+}(\vec{\chi}_{l_+^+}).c_{l_+^+} \rangle |$$

$$| \vdots |$$

$$| \mu \langle \mathfrak{s}_1^{\tau_1^m}(\vec{\chi}_1).c_1 | \dots | \mathfrak{s}_{l_m^m}^{\tau_m^m}(\vec{\chi}_{l_m^m}).c_{l_m^m} \rangle |$$

Positive stacks / evaluation contexts:

$$s_+, e_+ ::= \alpha^+ \mid \tilde{\mu}x^+.c$$

$$| \tilde{\mu} [\mathfrak{b}_1^{\tau_1^+}(\vec{\chi}_1).c_1 | \dots | \mathfrak{b}_{l_+^+}^{\tau_+^+}(\vec{\chi}_{l_+^+}).c_{l_+^+}] |$$

$$| \vdots |$$

$$| \tilde{\mu} [\mathfrak{b}_1^{\tau_1^n}(\vec{\chi}_1).c_1 | \dots | \mathfrak{b}_{l_n^n}^{\tau_n^n}(\vec{\chi}_{l_n^n}).c_{l_n^n}] |$$

Negative stacks:

$$s_- ::= \alpha^-$$

$$| \mathfrak{s}_1^{\tau_1^+}(\vec{a}) | \dots | \mathfrak{s}_{l_+^+}^{\tau_+^+}(\vec{a}) |$$

$$| \vdots | \quad | \cdot \vdots | \quad | \vdots |$$

$$| \mathfrak{s}_1^{\tau_1^m}(\vec{a}) | \dots | \mathfrak{s}_{l_m^m}^{\tau_m^m}(\vec{a}) |$$

Negative evaluation contexts:

$$e_- ::= \text{stk}^-(s_-) \mid \tilde{\mu}x^-.c$$

Commands:

$$c ::= \langle t_+ | e_+ \rangle^+ \mid \langle t_- | e_- \rangle^-$$

Figure V.1.3.b: Notations

Polarities:

$$\varepsilon ::= + \mid -$$

Arguments:

$$a ::= v_\varepsilon \mid s_\varepsilon$$

Variables:

$$\chi ::= x^\varepsilon \mid \alpha^\varepsilon$$

Term:

$$t ::= t_\varepsilon \mid v_\varepsilon \mid e_\varepsilon \mid s_\varepsilon \mid c$$

Figure V.1.3.c: Operational reduction

$$\begin{aligned}
 & \langle \mu \alpha^\varepsilon . c | s_\varepsilon \rangle^\varepsilon \triangleright_\mu c[s_\varepsilon / \alpha^\varepsilon] \\
 & \langle v_\varepsilon | \tilde{\mu} x^\varepsilon . c \rangle^\varepsilon \triangleright_{\tilde{\mu}} c[v_\varepsilon / x^\varepsilon] \\
 & \left\langle \mu \left\langle \mathfrak{s}_1^{\tau_1^j}(\vec{\chi}_1) . c_1 \mid \dots \mid \mathfrak{s}_l^{\tau_l^j}(\vec{\chi}_l) . c_l \right\rangle \left| \mathfrak{s}_k^{\tau_k^j}(\vec{a}) \right\rangle^- \right\rangle \triangleright_{\tau_-^j} c_k[\vec{a} / \vec{\chi}_k] \\
 & \left\langle \mathfrak{b}_k^{\tau_k^j}(\vec{a}) \left| \tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^j}(\vec{\chi}_1) . c_1 \mid \dots \mid \mathfrak{b}_l^{\tau_l^j}(\vec{\chi}_l) . c_l \right] \right\rangle^+ \right\rangle \triangleright_{\tau_+^j} c_k[\vec{a} / \vec{\chi}_k] \\
 & \triangleright \stackrel{\text{def}}{=} \triangleright_{\tilde{\mu}} \cup \triangleright_{\mu} \cup \left(\bigcup_j \triangleright_{\tau_-^j} \right) \cup \left(\bigcup_j \triangleright_{\tau_+^j} \right)
 \end{aligned}$$

 Figure V.1.3.d: Top-level η -expansion

$$\begin{aligned}
 & t_\varepsilon \stackrel{\eta}{\triangleright}_{d\mu} \mu \alpha^\varepsilon . \langle t_\varepsilon | \alpha^\varepsilon \rangle^\varepsilon && \text{if } \alpha^\varepsilon \text{ fresh w.r.t. } t_\varepsilon \\
 & e_\varepsilon \stackrel{\eta}{\triangleright}_{d\tilde{\mu}} \tilde{\mu} x^\varepsilon . \langle x^\varepsilon | e_\varepsilon \rangle^\varepsilon && \text{if } x^\varepsilon \text{ fresh w.r.t. } e_\varepsilon \\
 & v_- \stackrel{\eta}{\triangleright}_{d\tau_-^j} \mu \left\langle \mathfrak{s}_1^{\tau_1^j}(\vec{\chi}_1) . \left\langle v_- \left| \mathfrak{s}_1^{\tau_1^j}(\vec{\chi}_1) \right\rangle^- \right. \right. && \text{if } \vec{\chi}_1, \dots, \vec{\chi}_l \text{ fresh w.r.t. } v_- \\
 & \quad \vdots && \\
 & \left. \left. \mathfrak{s}_l^{\tau_l^j}(\vec{\chi}_l) . \left\langle v_- \left| \mathfrak{s}_l^{\tau_l^j}(\vec{\chi}_l) \right\rangle^- \right. \right\rangle && \\
 & s_+ \stackrel{\eta}{\triangleright}_{d\tau_+^j} \tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^j}(\vec{\chi}_1) . \left\langle \mathfrak{b}_1^{\tau_1^j}(\vec{\chi}_1) \left| s_+ \right\rangle^+ \right. \right. && \text{if } \vec{\chi}_1, \dots, \vec{\chi}_l \text{ fresh w.r.t. } s_+ \\
 & \quad \vdots && \\
 & \left. \left. \mathfrak{b}_l^{\tau_l^j}(\vec{\chi}_l) . \left\langle \mathfrak{b}_l^{\tau_l^j}(\vec{\chi}_l) \left| s_+ \right\rangle^+ \right. \right] && \\
 & \stackrel{\eta}{\triangleright} \stackrel{\text{def}}{=} \stackrel{\eta}{\triangleright}_{d\tilde{\mu}} \cup \stackrel{\eta}{\triangleright}_{d\mu} \cup \left(\bigcup_j \stackrel{\eta}{\triangleright}_{d\tau_-^j} \right) \cup \left(\bigcup_j \stackrel{\eta}{\triangleright}_{d\tau_+^j} \right)
 \end{aligned}$$

Figure V.1.4: The $L_p^{\rightarrow \Downarrow \Uparrow \neg_+ \neg_- \otimes \wp \oplus \& 1 \perp 0^\top}$ calculus

Figure V.1.4.a: Syntax

Positive values:

$$\begin{aligned} v_+, w_+ &::= x^+ \\ &| (v_+ \otimes w_+) \\ &| \iota_1(v_+) \mid \iota_2(v_+) \\ &| \{v_-\} \\ &| \neg_+(s_-) \\ &| () \end{aligned}$$

Positive expressions:

$$t_+, u_+ ::= \text{val}^+(v_+) \mid \mu \alpha^+.c$$

Negative values / expressions:

$$\begin{aligned} v_-, w_-, t_-, u_- &::= x^- \mid \mu \alpha^-.c \\ &| \mu(x^+ \cdot \alpha^-).c \\ &| \mu(\alpha^- \wp \beta^-).c \\ &| \mu \langle (\pi_1 \cdot \alpha_1^-).c_1 \mid (\pi_2 \cdot \alpha_2^-).c_2 \rangle \\ &| \mu \{\alpha^+\}.c \\ &| \mu \neg_-(x^+).xc \\ &| \mu \tilde{()}.c \\ &| \mu \langle \rangle \end{aligned}$$

Positive stacks / evaluation contexts:

$$\begin{aligned} s_+, e_+ &::= \alpha^+ \mid \tilde{\mu} x^+.c \\ &| \tilde{\mu}(x^+ \otimes y^+).c \\ &| \tilde{\mu}[\iota_1(x_1^+).c_1 \mid \iota_2(x_2^+).c_2] \\ &| \tilde{\mu}\{x^-\}.c \\ &| \tilde{\mu} \neg_+(\alpha^-). \alpha c \\ &| \tilde{\mu}().c \\ &| \tilde{\mu}[] \end{aligned}$$

Negative stacks:

$$\begin{aligned} s_- &::= \alpha^- \\ &| v_+ \cdot s_- \\ &| (s_-^1 \wp s_-^2) \\ &| \pi_1 \cdot s_- \mid \pi_2 \cdot s_- \\ &| \{s_+\} \\ &| \neg_-(v_+) \\ &| \tilde{()} \end{aligned}$$

Negative evaluation contexts:

$$e_- ::= \text{stk}^-(s_-) \mid \tilde{\mu} x^-.c$$

Commands:

$$c ::= \langle t_+ \mid e_+ \rangle^+ \mid \langle t_- \mid e_- \rangle^-$$

Figure V.1.4.b: Operational reduction

$$\begin{aligned}
& \langle \mu \alpha^\varepsilon . c \mid s_\varepsilon \rangle^\varepsilon \triangleright_\mu c[s_\varepsilon / \alpha^\varepsilon] \\
& \langle v_\varepsilon \mid \tilde{\mu} x^\varepsilon . c \rangle^\varepsilon \triangleright_{\tilde{\mu}} c[v_\varepsilon / x^\varepsilon] \\
& \langle \mu(x^+ \cdot \alpha^-) . c \mid v_+ \cdot s_- \rangle^- \triangleright_{\rightarrow} c[v_+ / x^+, s_- / \alpha^-] \\
& \langle \mu\{\alpha^+\} . c \mid \{s_+\} \rangle^- \triangleright_{\uparrow} c[s_+ / \alpha^+] \\
& \langle \mu\neg_-(x^+) . xc \mid \neg_-(v_+) \rangle^- \triangleright_{\neg_-} c[v_+ / x^+] \\
& \langle \mu(\alpha^- \wp \beta^-) . c \mid (s_-^1 \wp s_-^2) \rangle^- \triangleright_{\wp} c[s_-^1 / \alpha^-, s_-^2 / \beta^-] \\
& \langle \mu \langle (\pi_1 \cdot \alpha_1^-) . c^1 \mid (\pi_2 \cdot \alpha_2^-) . c^2 \rangle \mid \pi_i \cdot s_- \rangle^- \triangleright_{\&} c^i[s_- / \alpha_i^-] \\
& \langle \mu \tilde{()}. c \mid \tilde{()} \rangle^- \triangleright_{\perp} c \\
& (\triangleright_{\top} \text{ is trivial}) \\
& \langle \{v_-\} \mid \tilde{\mu}\{x^-\} . c \rangle^+ \triangleright_{\downarrow} c[v_- / x^-] \\
& \langle \neg_+(s_-) \mid \tilde{\mu}\neg_+(\alpha^-) . \alpha c \rangle^+ \triangleright_{\neg_+} c[s_- / \alpha^-] \\
& \langle (v_+ \otimes w_+) \mid \tilde{\mu}(x^+ \otimes y^+) . c \rangle^+ \triangleright_{\otimes} c[v_+ / x^+, w_+ / y^+] \\
& \langle \iota_i(v_+) \mid \tilde{\mu}[\iota_1(x_1^+) . c^1 \mid \iota_2(x_2^+) . c^2] \rangle^+ \triangleright_{\oplus} c^i[v_+ / x_i^+] \\
& \langle () \mid \tilde{\mu}(). c \rangle^+ \triangleright_1 c \\
& (\triangleright_0 \text{ is trivial})
\end{aligned}$$

$$\triangleright \stackrel{\text{def}}{=} \triangleright_{\tilde{\mu}} \cup \triangleright_{\tilde{\mu}} \cup \triangleright_{\rightarrow} \cup \triangleright_{\wp} \cup \triangleright_{\&} \cup \triangleright_{\uparrow} \cup \triangleright_{\neg_-} \cup \triangleright_{\perp} \cup \triangleright_{\otimes} \cup \triangleright_{\oplus} \cup \triangleright_{\downarrow} \cup \triangleright_{\neg_+} \cup \triangleright_1$$

Figure V.1.4.c: Top-level η -expansion

$t_\varepsilon \stackrel{q}{\rightarrow}_{d\mu} \mu\alpha^\varepsilon.\langle t_\varepsilon \alpha^\varepsilon \rangle^\varepsilon$	if α^ε fresh w.r.t. t_ε
$e_\varepsilon \stackrel{q}{\rightarrow}_{d\tilde{\mu}} \tilde{\mu}x^\varepsilon.\langle x^\varepsilon e_\varepsilon \rangle^\varepsilon$	if x^ε fresh w.r.t. e_ε
$v_- \stackrel{q}{\rightarrow}_{d\rightarrow} \mu(x^+ \cdot \alpha^-).\langle v_- x^+ \cdot \alpha^- \rangle^-$	if x^+ and α^- fresh w.r.t. v_-
$v_- \stackrel{q}{\rightarrow}_{d\uparrow} \mu\{\alpha^+\}.\langle v_- \{\alpha^+\} \rangle^-$	if α^+ fresh w.r.t. v_-
$v_- \stackrel{q}{\rightarrow}_{d\neg} \mu\neg_-(x^+).\mathbf{x}\langle v_- \neg_-(x^+) \rangle^-$	if x^+ fresh w.r.t. v_-
$v_- \stackrel{q}{\rightarrow}_{d\mathfrak{A}} \mu(\alpha^- \mathfrak{A} \beta^-).\langle v_- (\alpha^- \mathfrak{A} \beta^-) \rangle^-$	if α^- and β^- fresh w.r.t. v_-
$v_- \stackrel{q}{\rightarrow}_{d\&} \mu\langle (\pi_1 \cdot \alpha_1^-).\langle v_- \alpha_1^- \rangle^- (\pi_2 \cdot \alpha_2^-).\langle v_- \alpha_2^- \rangle^- \rangle$	if α_1^- and α_2^- fresh w.r.t. v_-
$v_- \stackrel{q}{\rightarrow}_{d\perp} \mu\tilde{()}. \langle v_- \tilde{()} \rangle^-$	
$v_- \stackrel{q}{\rightarrow}_{d\top} \mu\langle \rangle$	
$s_+ \stackrel{q}{\rightarrow}_{d\downarrow} \tilde{\mu}\{x^-\}.\langle \{x^-\} s_+ \rangle^+$	if x^- fresh w.r.t. s_+
$s_+ \stackrel{q}{\rightarrow}_{d\neg_+} \tilde{\mu}\neg_+(\alpha^-).\alpha\langle \neg_+(\alpha^-) s_+ \rangle^+$	if α^- fresh w.r.t. s_+
$s_+ \stackrel{q}{\rightarrow}_{d\otimes} \tilde{\mu}(x^+ \otimes y^+).\langle (x^+ \otimes y^+) s_+ \rangle^+$	if x^+ and y^+ fresh w.r.t. s_+
$s_+ \stackrel{q}{\rightarrow}_{d\oplus} \tilde{\mu}[\iota_1(x_1^+).\langle \iota_1(x_1^+) s_+ \rangle^+ \iota_2(x_2^+).\langle \iota_2(x_2^+) s_+ \rangle^+]$	if x_1^+ and x_2^+ fresh w.r.t. s_+
$s_+ \stackrel{q}{\rightarrow}_{d1} \tilde{\mu}().\langle () s_+ \rangle^+$	
$s_+ \stackrel{q}{\rightarrow}_{d0} \tilde{\mu}[]$	

$$\stackrel{q}{\rightarrow} \stackrel{\text{def}}{=} \stackrel{q}{\rightarrow}_{d\tilde{\mu}} \cup \stackrel{q}{\rightarrow}_{d\mu} \cup \stackrel{q}{\rightarrow}_{d\rightarrow} \cup \stackrel{q}{\rightarrow}_{d\uparrow} \cup \stackrel{q}{\rightarrow}_{d\neg} \cup \stackrel{q}{\rightarrow}_{d\mathfrak{A}} \cup \stackrel{q}{\rightarrow}_{d\&} \cup \stackrel{q}{\rightarrow}_{d\perp} \cup \stackrel{q}{\rightarrow}_{d\top} \cup \stackrel{q}{\rightarrow}_{d\downarrow} \cup \stackrel{q}{\rightarrow}_{d\neg_+} \cup \stackrel{q}{\rightarrow}_{d\otimes} \cup \stackrel{q}{\rightarrow}_{d\oplus} \cup \stackrel{q}{\rightarrow}_{d1} \cup \stackrel{q}{\rightarrow}_{d0}$$

Remark V.1.2


Note that the coercions val^+ (resp. stk^-) are what ensures that the syntax is closed under η -expansions. Indeed, if we removed val^+ (resp. stk^-), then we would have

$$v_+ \stackrel{\dagger}{\rightarrow}_{\mu} \mu\alpha^+.\langle v_+|\alpha^+ \rangle^+ \quad (\text{resp. } s_- \stackrel{\dagger}{\rightarrow}_{\bar{\mu}} \bar{\mu}x^+.\langle x^-|s_- \rangle^-)$$

and hence

$$(v_+ \otimes w_+) \stackrel{\dagger}{\rightarrow} ((\mu\alpha^+.\langle v_+|\alpha^+ \rangle^+) \otimes w_+) \quad (\text{resp. } v_+ \cdot s_- \stackrel{\dagger}{\rightarrow} v_+ \cdot (\bar{\mu}x^+.\langle x^-|s_- \rangle^-))$$

With the coercions, this problem disappears because v_+ (resp. s_-) can not be η -expanded on its own, and $(\text{val}^+(v_+) \otimes w_+)$ (resp. $v_+ \cdot \text{stk}^-(s_-)$) is not within the syntax^a.

^aOf course, one can also fix this by allowing expressions (resp. evaluation contexts) in value and stack constructors, see .

Remark V.1.3

We could also add coercions val^- (resp. stk^+), i.e. define negative expressions (resp. positive evaluation contexts) by

$$t_-, u_- ::= \text{val}^-(v_-) \quad (\text{resp. } e_+ ::= \text{stk}^+(s_+))$$

While this could be useful in future calculi, here it would be completely superfluous (because these coercions would be bijections), while requiring duplications in the definition of $\stackrel{\dagger}{\rightarrow}$:

$$t_- \stackrel{\dagger}{\rightarrow}_{\mu} \mu\alpha^+.\langle t_-|\alpha^+ \rangle^+ \quad (\text{resp. } e_+ \stackrel{\dagger}{\rightarrow}_{\bar{\mu}} \bar{\mu}x^+.\langle x^+|e_+ \rangle^+)$$

would need to be replaced by

$$v_- \stackrel{\dagger}{\rightarrow}_{\mu} \mu\alpha^+.\langle v_-|\alpha^+ \rangle^+ \quad (\text{resp. } s_+ \stackrel{\dagger}{\rightarrow}_{\bar{\mu}} \bar{\mu}x^+.\langle x^+|s_+ \rangle^+)$$

to ensure that e.g.

$$\{v_-\} \stackrel{\dagger}{\rightarrow}_{\mu} \{\mu\alpha^+.\langle v_-|\alpha^+ \rangle^+\} \quad (\text{resp. } \{s_+\} \stackrel{\dagger}{\rightarrow}_{\bar{\mu}} \{\bar{\mu}x^+.\langle x^+|s_+ \rangle^+\})$$

while

$$t_+ \stackrel{\dagger}{\rightarrow}_{\mu} \mu\alpha^+.\langle t_+|\alpha^+ \rangle^+ \quad (\text{resp. } e_- \stackrel{\dagger}{\rightarrow}_{\bar{\mu}} \bar{\mu}x^+.\langle x^+|e_- \rangle^+)$$

can not be made to act on v_+ (resp. s_-) if one wants η -expansion to preserve the syntax.

Normal forms, clashes and waiting commands

There are several kinds of \triangleright -normal forms:

Definition V.1.4

A command c is said to be:

- a *clash* when it is of one of the following shapes:

$$c = \left\langle \mathbf{b}_k^{\tau_+^{j_1}}(\vec{a}) \middle| \mu \left[\mathbf{b}_1^{\tau_+^{j_2}}(\vec{\chi}_1).c_1 \mid \dots \mid \mathbf{b}_l^{\tau_+^{j_2}}(\vec{\chi}_l).c_l \right] \right\rangle^+ \text{ with } \tau_+^{j_1} \neq \tau_+^{j_2}, \text{ or}$$

$$c = \left\langle \mu \left\langle \mathbf{s}_1^{\tau_-^{j_1}}(\vec{\chi}_1).c_1 \mid \dots \mid \mathbf{s}_l^{\tau_-^{j_1}}(\vec{\chi}_l).c_l \right\rangle \middle| \mathbf{s}_k^{\tau_-^{j_2}}(\vec{a}) \right\rangle^- \text{ with } \tau_-^{j_1} \neq \tau_-^{j_2}$$

- *waiting* when it is of one of the following shapes:

$$c = \langle x^\varepsilon \mid \alpha^\varepsilon \rangle^\varepsilon, \quad c = \left\langle \mathbf{b}_k^{\tau_+^j}(\vec{a}) \middle| \alpha^+ \right\rangle^+, \quad c = \left\langle x^+ \middle| \mu \left[\mathbf{b}_1^{\tau_+^j}(\vec{\chi}_1).c_1 \mid \dots \mid \mathbf{b}_l^{\tau_+^j}(\vec{\chi}_l).c_l \right] \right\rangle^+,$$

$$c = \left\langle x^- \middle| \mathbf{s}_k^{\tau_-^j}(\vec{a}) \right\rangle^-, \quad \text{or} \quad c = \left\langle \mu \left\langle \mathbf{s}_1^{\tau_-^j}(\vec{\chi}_1).c_1 \mid \dots \mid \mathbf{s}_l^{\tau_-^j}(\vec{\chi}_l).c_l \right\rangle \middle| \alpha^- \right\rangle^-$$

Example V.1.5

The commands

$$\langle \iota_1(v_+) \mid \tilde{\mu}(x^+ \otimes y^+).c \rangle^+ \quad \text{and} \quad \left\langle \mathbf{b}_k^{\tau_+^j}(\vec{a}) \middle| \tilde{\mu}[] \right\rangle^+$$

are clashes.

These two definitions cover exactly all \triangleright -normal commands:

Fact V.1.6

A command is \triangleright -normal if and only if it is either a clash or a waiting command, and those two cases are mutually exclusive.

Proof

By case analysis on the command.

We now look at the effect of disubstitutions on \triangleright -normal commands. For clashes, disubstitutions have no effect:

Fact V.1.7

The set of clashes is disubstitutive: for any clash c and disubstitution φ , the command $c[\varphi]$ is a clash.

Proof

By case analysis on c .

Waiting commands are waiting on a particular variable, and until this variable is substituted by a non-variable, they remain waiting:

Definition V.1.8

A command c is said to be:

- waiting for x^- if it is of the shape $c = \langle x^- | \alpha^- \rangle^-$ or $c = \langle x^- | \mathfrak{s}_k^{\tau_k^j}(\vec{a}) \rangle^-$;
- waiting for α^+ if it is of the shape $c = \langle x^+ | \alpha^+ \rangle^+$ or $c = \langle \mathfrak{b}_k^{\tau_k^j}(\vec{a}) | \alpha^+ \rangle^+$;
- waiting for α^- if it is of the shape $c = \langle \mu \langle \mathfrak{s}_1^{\tau_1^j}(\vec{\chi}_1) . c_1 | \dots | \mathfrak{s}_l^{\tau_l^j}(\vec{\chi}_l) . c_l \rangle | \alpha^- \rangle^-$;
- waiting for x^+ if it is of the shape $c = \langle x^+ | \tilde{\mu} [\mathfrak{b}_1^{\tau_1^j}(\vec{\chi}_1) . c_1 | \dots | \mathfrak{b}_l^{\tau_l^j}(\vec{\chi}_l) . c_l] \rangle^+$.

Fact V.1.9

If c is waiting for χ then there exists an argument a such that $c[a/\chi]$ is reducible.

Proof

Defining a as $\mu \star^-.c$, $\tilde{\mu} x^+.c$, $\mathfrak{s}_k^{\tau_k^j}(\vec{a})$ or $\mathfrak{b}_k^{\tau_k^j}(\vec{a})$ works when χ is x^- , α^+ , α^- and x^+ respectively (with τ_ϵ^j being the type former of the displayed $\mu \langle \dots \rangle$ or $\tilde{\mu} [\dots]$).

Fact V.1.10

If c is waiting for χ then for any disubstitution φ such that $\varphi(\chi)$ is a variable, $c[\varphi]$ is waiting for $\varphi(\chi)$.

Proof

By case analysis on c .

Remark V.1.11

If c is waiting for χ then given a non-variable a , $c[a/\chi]$ may be:

- reducible, e.g.

$$\langle \mathfrak{b}_k^{\tau_k^j}(\vec{a}) | \alpha^+ \rangle^+ [\tilde{\mu} x^+.c / \alpha^+] = \langle \mathfrak{b}_k^{\tau_k^j}(\vec{a}) | \tilde{\mu} x^+.c \rangle^+$$

is reducible;

- a clash, e.g.

$$\langle \mathfrak{b}_k^{\tau_k^j}(\vec{a}) | \alpha^+ \rangle^+ [\tilde{\mu} [\mathfrak{b}_1^{\tau_1^j}(\vec{\chi}_1) . c_1 | \dots | \mathfrak{b}_l^{\tau_l^j}(\vec{\chi}_l) . c_l] / \alpha^+]$$

is a clash when $j_1 \neq j_2$;

- waiting for another variable, e.g. $\langle x^- | \alpha^- \rangle^-$ is waiting for x^- and

$$\langle x^- | \alpha^- \rangle^- [\mu(y^+ \cdot \beta^-).c/x^-] = \langle \mu(y^+ \cdot \beta^-).c | \alpha^- \rangle^-$$

is waiting for α^- .

This last case could make us want to say that $\langle x^- | \alpha^- \rangle^-$ waits for both x^- and α^- but this is not really the case because

$$\langle x^- | \alpha^- \rangle^- [\mu\beta^-.c/x^-] = \langle \mu\beta^-.c | \alpha^- \rangle^-$$

is not waiting for α^- in general (e.g. it \triangleright -diverges whenever c does).

Definition V.1.12

A command c is said to:

- *converge to c'* , written $c \Downarrow c'$ or $c \triangleright^\oplus c'$, when $c \triangleright^* c' \Downarrow$;
- *converge*, written $c \Downarrow$ or $c \triangleright^\oplus$, when there exists some c' it converges to;
- *diverge*, written $c \Uparrow$ or $c \triangleright^\omega$, when there is an infinite reduction sequence

$$c \triangleright c' \triangleright c'' \triangleright \dots$$

starting at c .

There are three possible outcomes for a command:

Fact V.1.13

Any command either diverges, converges to a clash, or converges to a waiting command, and those three cases are mutually exclusive.

Proof

By determinism of \triangleright , it either converges or diverges, and by Fact V.1.6, the \triangleright -normal command it converges to is either a clash or a waiting command.

Properties

Just like we distinguish substitution from disubstitutions, we distinguish substitutivity from disubstitutivity:

Definition V.1.14

A reduction \rightsquigarrow of $L_p^{\vec{\tau}}$ is said to be *substitutive* (resp. *disubstitutive*) when for any terms t and \vec{t} , and substitution σ (resp. disubstitution φ), we have

$$t \rightsquigarrow t' \Rightarrow t[\sigma] \rightsquigarrow t'[\sigma] \quad (\text{resp. } t \rightsquigarrow t' \Rightarrow t[\varphi] \rightsquigarrow t'[\varphi])$$

The properties of the reductions are summarized in Figure V.1

 Table V.1.: Properties of reductions in the $L_p^{\vec{\tau}}$ calculus

	\triangleright	\rightarrow	$\overset{\circ}{\triangleright}$	\dashv
Substitutive	✓	✓	✓	✓
Disubstitutive	✓	✓	✓	✓
Deterministic	✓	✗	✗	✗
Confluent	✓	✓	✓	✓
Postpones after \triangleright	✓	✓	✓	✓

The proofs of all of these properties are either trivial or routine, and are therefore relegated to [A](#) in the appendix. Confluence and postponement are proven in a standard way using a parallel reduction \Rightarrow [Tak95; Bar84]. The only slightly non-standard choice is the definition of \Rightarrow :

Remark V.1.15

The most common definitions of the parallel reduction \Rightarrow contain rules two kinds of rules: those that simply combine reduction sequences on subterms such as

$$\frac{t_\varepsilon \Rightarrow t'_\varepsilon \quad e_\varepsilon \Rightarrow e'_\varepsilon}{\langle t_\varepsilon | e_\varepsilon \rangle^\varepsilon \Rightarrow \langle t'_\varepsilon | e'_\varepsilon \rangle^\varepsilon}, \quad \frac{v_+ \Rightarrow v'_+}{l_i(v_+) \Rightarrow l_i(v'_+)}, \quad \text{and} \quad \frac{c_1 \Rightarrow c'_1 \quad c_2 \Rightarrow c'_2}{\tilde{\mu} \left[\begin{array}{l} l_1(x_1^+) \cdot c_1 \\ l_2(x_2^+) \cdot c_2 \end{array} \right] \Rightarrow \tilde{\mu} \left[\begin{array}{l} l_1(x_1^+) \cdot c'_1 \\ l_2(x_2^+) \cdot c'_2 \end{array} \right]}$$

and those that add a reduction step such as

$$\frac{v_+ \Rightarrow v'_+ \quad c_1 \Rightarrow c'_1 \quad c_2 \Rightarrow c'_2}{\left\langle l_i(v_+) \left| \tilde{\mu} \left[\begin{array}{l} l_1(x_1^+) \cdot c_1 \\ l_2(x_2^+) \cdot c_2 \end{array} \right] \right. \right\rangle^+ \Rightarrow c'_i[v'_+/x_i^+]}$$

Let \Rightarrow be the restriction of \Rightarrow defined by $t \Rightarrow t'$ meaning that there exists a derivation of $t \Rightarrow t'$ whose last rule is not a step rule. With the usual definition of \Rightarrow , the

[Tak95] “Parallel Reductions in λ -Calculus”, Takahashi, 1995

[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

two following rules are admissible:

$$\frac{t \Rightarrow t'}{t \Rightarrow t'} \quad \text{and} \quad \frac{t \Rightarrow t' \quad t' \triangleright t''}{t \Rightarrow t''}$$

For $L_p^{\bar{\tau}}$, since there are four kinds of \triangleright reductions (namely \triangleright_μ , $\triangleright_{\bar{\mu}}$, $\triangleright_{\tau_-^j}$, and $\triangleright_{\tau_+^j}$), the usual definition of \Rightarrow has four step rules. It is hence slightly easier to define \Rightarrow and \Rightarrow by mutual induction by adding the two rules above in the definition, removing the step rules, and strengthening the other rules to remember that no step was taken at the top-level, e.g.

$$\frac{t_\varepsilon \Rightarrow t'_\varepsilon \quad e_\varepsilon \Rightarrow e'_\varepsilon}{\langle t_\varepsilon | e_\varepsilon \rangle^\varepsilon \Rightarrow \langle t'_\varepsilon | e'_\varepsilon \rangle^\varepsilon}, \quad \frac{v_+ \Rightarrow v'_+}{l_i(v_+) \Rightarrow l_i(v'_+)}, \quad \text{and} \quad \frac{c_1 \Rightarrow c'_1 \quad c_2 \Rightarrow c'_2}{\begin{array}{c} \bar{\mu}[l_1(x_1^+).c_1] \\ [l_2(x_2^+).c_2] \end{array} \Rightarrow \begin{array}{c} \bar{\mu}[l_1(x_1^+).c'_1] \\ [l_2(x_2^+).c'_2] \end{array}}$$

The usual step rules are then derivable, e.g.

$$\frac{\frac{v_+ \Rightarrow v'_+}{l_i(v_+) \Rightarrow l_i(v'_+)}, \quad \frac{\frac{c_1 \Rightarrow c'_1 \quad c_2 \Rightarrow c'_2}{\begin{array}{c} \bar{\mu}[l_1(x_1^+).c_1] \\ [l_2(x_2^+).c_2] \end{array} \Rightarrow \begin{array}{c} \bar{\mu}[l_1(x_1^+).c'_1] \\ [l_2(x_2^+).c'_2] \end{array}}{\begin{array}{c} \bar{\mu}[l_1(x_1^+).c_1] \\ [l_2(x_2^+).c_2] \end{array} \Rightarrow \begin{array}{c} \bar{\mu}[l_1(x_1^+).c'_1] \\ [l_2(x_2^+).c'_2] \end{array}}}{\frac{\left\langle l_i(v_+) \left| \begin{array}{c} \bar{\mu}[l_1(x_1^+).c_1] \\ [l_2(x_2^+).c_2] \end{array} \right. \right\rangle^+ \Rightarrow \left\langle l_i(v'_+) \left| \begin{array}{c} \bar{\mu}[l_1(x_1^+).c'_1] \\ [l_2(x_2^+).c'_2] \end{array} \right. \right\rangle^+}{\left\langle l_i(v_+) \left| \begin{array}{c} \bar{\mu}[l_1(x_1^+).c_1] \\ [l_2(x_2^+).c_2] \end{array} \right. \right\rangle^+ \Rightarrow \left\langle l_i(v'_+) \left| \begin{array}{c} \bar{\mu}[l_1(x_1^+).c'_1] \\ [l_2(x_2^+).c'_2] \end{array} \right. \right\rangle^+ \triangleright c'_i[v'_+/x_i^+]}$$

This alternative definition of \Rightarrow also has the advantage of disentangling the part of \Rightarrow that depends on \triangleright from the rest, which makes \Rightarrow parametric in \triangleright .

V.1.3. Well-typed and well-polarized terms

Well-typed terms

Simply typed $L_p^{\bar{\tau}}$ is described in Figure V.1.5. Each type former τ_+^j (resp. τ_-^j) has logic rules $(\vdash_{\mathbf{b}_k^{\tau_+^j}})$ (resp. $(\mathbf{s}_k^{\tau_-^j} \vdash)$) that introduce its constructors, and $(\tau_+^j \vdash)$ (resp. $(\vdash \tau_-^j)$) that introduces the correspond pattern match. Of course, in these logic rules, sequence of types given to τ_ε^j in the conclusions depends on the type in the premises, and if one wants the subformula property to hold, one should require $\{\vec{A}\} \subseteq \{\vec{B}\}$ in $(\mathbf{s}_k^{\tau_-^j} \vdash)$ and $(\vdash_{\mathbf{b}_k^{\tau_+^j}})$ and $\{\vec{A}, \vec{B}\} \subseteq \{\vec{C}\}$ in $(\vdash \tau_-^j)$ and $(\tau_+^j \vdash)$.

Figure V.1.5: Simply typed $L_p^{\bar{\tau}}$

Figure V.1.5.a: Core rules

$$\begin{array}{c}
 \frac{}{x^\varepsilon : A_\varepsilon \vdash \underline{x^\varepsilon : A_\varepsilon} \mid} \text{ (}\vdash\text{AX)} \qquad \frac{}{\mid \underline{\alpha^\varepsilon : A_\varepsilon} \vdash \alpha^\varepsilon : A_\varepsilon} \text{ (AX}\vdash\text{)} \\
 \\
 \frac{c : (\Gamma \vdash \alpha^\varepsilon : A_\varepsilon, \Delta)}{\Gamma \vdash \underline{\mu \alpha^\varepsilon . c : A_\varepsilon} \mid \Delta} \text{ (}\vdash\mu\text{)} \qquad \frac{c : (\Gamma, x^\varepsilon : A_\varepsilon \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu} x^\varepsilon . c : A_\varepsilon} \vdash \Delta} \text{ (}\tilde{\mu}\vdash\text{)} \\
 \\
 \frac{\Gamma_1 \vdash \underline{t_\varepsilon : A_\varepsilon} \mid \Delta_1 \quad \Gamma_2 \mid \underline{e_\varepsilon : A_\varepsilon} \vdash \Delta_2}{\langle \underline{t_\varepsilon \mid e_\varepsilon} \rangle^\varepsilon : (\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2)} \text{ (CUT)}
 \end{array}$$

Figure V.1.5.b: Structural rules (commands)

$$\begin{array}{c}
 \frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma \vdash \alpha^\varepsilon : A_\varepsilon, \Delta)} \text{ (}\vdash\text{wc)} \qquad \frac{c : (\Gamma \vdash \alpha_1^\varepsilon : A_\varepsilon, \alpha_2^\varepsilon : A_\varepsilon, \Delta)}{c[\beta^\varepsilon / \alpha_1^\varepsilon, \beta^\varepsilon / \alpha_2^\varepsilon] : (\Gamma \vdash \beta^\varepsilon : A_\varepsilon, \Delta)} \text{ (}\vdash\text{cc)} \\
 \\
 \frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma, x^\varepsilon : A_\varepsilon \vdash \Delta)} \text{ (wc}\vdash\text{)} \qquad \frac{c : (\Gamma, x_1^\varepsilon : A_\varepsilon, x_2^\varepsilon : A_\varepsilon \vdash \Delta)}{c[y^\varepsilon / x_1^\varepsilon, y^\varepsilon / x_2^\varepsilon] : (\Gamma, y^\varepsilon : A_\varepsilon \vdash \Delta)} \text{ (cc}\vdash\text{)} \\
 \\
 \frac{c : (\Gamma \vdash \Delta_1, \alpha_1^\varepsilon : A_\varepsilon, \alpha_2^\varepsilon : A_\varepsilon, \Delta_2)}{c : (\Gamma \vdash \Delta_1, \alpha_2^\varepsilon : A_\varepsilon, \alpha_1^\varepsilon : A_\varepsilon, \Delta_2)} \text{ (}\vdash\text{PC)} \qquad \frac{c : (\Gamma_1, x_1^\varepsilon : A_\varepsilon, x_2^\varepsilon : A_\varepsilon, \Gamma_2 \vdash \Delta)}{c : (\Gamma_1, x_2^\varepsilon : A_\varepsilon, x_1^\varepsilon : A_\varepsilon, \Gamma_2 \vdash \Delta)} \text{ (PC}\vdash\text{)}
 \end{array}$$

Figure V.1.5.c: Structural rules (expressions)

$$\begin{array}{c}
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta}{\Gamma \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \alpha^\varepsilon : B_\varepsilon, \Delta} \text{ (}\vdash\text{wt)} \qquad \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \alpha_1^\varepsilon : B_\varepsilon, \alpha_2^\varepsilon : B_\varepsilon, \Delta}{\Gamma \vdash \underline{t_{\varepsilon_0} [\beta^\varepsilon / \alpha_1^\varepsilon, \beta^\varepsilon / \alpha_2^\varepsilon] : A_{\varepsilon_0}} \mid \beta^\varepsilon : B_\varepsilon, \Delta} \text{ (}\vdash\text{ct)} \\
 \\
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta}{\Gamma, x^\varepsilon : B_\varepsilon \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta} \text{ (wt}\vdash\text{)} \qquad \frac{\Gamma, x_1^\varepsilon : B_\varepsilon, x_2^\varepsilon : B_\varepsilon \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta}{\Gamma, x^\varepsilon : B_\varepsilon \vdash \underline{t_{\varepsilon_0} [x^\varepsilon / x_1^\varepsilon, x^\varepsilon / x_2^\varepsilon] : A_{\varepsilon_0}} \mid \Delta} \text{ (ct}\vdash\text{)} \\
 \\
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta_1, \alpha_1^\varepsilon : B_\varepsilon, \alpha_2^\varepsilon : B_\varepsilon, \Delta_2}{\Gamma \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta_1, \alpha_2^\varepsilon : B_\varepsilon, \alpha_1^\varepsilon : B_\varepsilon, \Delta_2} \text{ (}\vdash\text{Pt)} \qquad \frac{\Gamma_1, x_1^\varepsilon : B_\varepsilon, x_2^\varepsilon : B_\varepsilon, \Gamma_2 \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta}{\Gamma_1, x_2^\varepsilon : B_\varepsilon, x_1^\varepsilon : B_\varepsilon, \Gamma_2 \vdash \underline{t_{\varepsilon_0} : A_{\varepsilon_0}} \mid \Delta} \text{ (Pt}\vdash\text{)}
 \end{array}$$

Figure V.1.5.d: Structural rules (evaluation contexts)

$$\begin{array}{c}
 \frac{\Gamma \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta}{\Gamma \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \alpha^\varepsilon : B_\varepsilon, \Delta} \text{ (}\vdash\text{we)} \quad \frac{\Gamma \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \alpha_1^\varepsilon : B_\varepsilon, \alpha_2^\varepsilon : B_\varepsilon, \Delta}{\Gamma \mid \underline{e_{\varepsilon_0} [\beta^\varepsilon / \alpha_1^\varepsilon, \beta^\varepsilon / \alpha_2^\varepsilon] : A_{\varepsilon_0}} \vdash \beta^\varepsilon : B_\varepsilon, \Delta} \text{ (}\vdash\text{ce)} \\
 \\
 \frac{\Gamma \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta}{\Gamma, x^\varepsilon : B_\varepsilon \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta} \text{ (we}\vdash\text{)} \quad \frac{\Gamma, x_1^\varepsilon : B_\varepsilon, x_2^\varepsilon : B_\varepsilon \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta}{\Gamma, x^\varepsilon : B_\varepsilon \mid \underline{e_{\varepsilon_0} [x^\varepsilon / x_1^\varepsilon, x^\varepsilon / x_2^\varepsilon] : A_{\varepsilon_0}} \vdash \Delta} \text{ (ce}\vdash\text{)} \\
 \\
 \frac{\Gamma \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta_1, \alpha_1^\varepsilon : B_\varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta_2}{\Gamma \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta_1, \alpha_2^\varepsilon : B_\varepsilon, \alpha_1^\varepsilon : \varepsilon, \Delta_2} \text{ (}\vdash\text{pe)} \quad \frac{\Gamma_1, x_1^\varepsilon : B_\varepsilon, x_2^\varepsilon : B_\varepsilon, \Gamma_2 \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta}{\Gamma_1, x_2^\varepsilon : B_\varepsilon, x_1^\varepsilon : B_\varepsilon, \Gamma_2 \mid \underline{e_{\varepsilon_0} : A_{\varepsilon_0}} \vdash \Delta} \text{ (pe}\vdash\text{)}
 \end{array}$$

Figure V.1.5.e: General shape of logic rules

$$\begin{array}{c}
 \frac{\Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : A_{\varepsilon_1}^1} \mid \Delta_1 \quad \dots \quad \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : A_{\varepsilon_q}^q} \mid \Delta_q \quad \Gamma_{q+1} \mid \underline{s_{\varepsilon_{q+1}}^1 : A_{\varepsilon_{q+1}}^{q+1}} \vdash \Delta_{q+1} \quad \dots \quad \Gamma_{q+r} \mid \underline{s_{\varepsilon_{q+r}}^r : A_{\varepsilon_{q+r}}^{q+r}} \vdash \Delta_{q+r}}{\Gamma_1, \dots, \Gamma_{q+r} \mid \underline{\mathfrak{s}_k^{\tau_-^j} (v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1, \dots, s_{\varepsilon_{q+r}}^r) : \tau_-^j(\vec{B})} \vdash \Delta_1, \dots, \Delta_{q+r}} \left(\mathfrak{s}_k^{\tau_-^j} \vdash \right) \\
 \\
 \frac{c_1 : (\Gamma, \vec{x}_1 : \vec{A}^1 \vdash \vec{\alpha}_1 : \vec{B}^1, \Delta) \quad \dots \quad c_l : (\Gamma, \vec{x}_l : \vec{A}^l \vdash \vec{\alpha}_l : \vec{B}^l, \Delta)}{\Gamma \vdash \underline{\mu \langle \mathfrak{s}_1^{\tau_-^j}(\vec{x}_1, \vec{\alpha}_1).c_1 \mid \dots \mid \mathfrak{s}_l^{\tau_-^j}(\vec{x}_l, \vec{\alpha}_l).c_l \rangle : \tau_-^j(\vec{C})} \mid \Delta} \text{ (}\vdash\tau_-^j\text{)} \\
 \\
 \frac{\Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : A_{\varepsilon_1}^1} \mid \Delta_1 \quad \dots \quad \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : A_{\varepsilon_q}^q} \mid \Delta_q \quad \Gamma_{q+1} \mid \underline{s_{\varepsilon_{q+1}}^1 : A_{\varepsilon_{q+1}}^{q+1}} \vdash \Delta_{q+1} \quad \dots \quad \Gamma_{q+r} \mid \underline{s_{\varepsilon_{q+r}}^r : A_{\varepsilon_{q+r}}^{q+r}} \vdash \Delta_{q+r}}{\Gamma_1, \dots, \Gamma_q \vdash \underline{\mathfrak{b}_k^{\tau_+^j} (v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1, \dots, s_{\varepsilon_{q+r}}^r) : \tau_+^j(\vec{B})} \mid \Delta_1, \dots, \Delta_q} \left(\vdash \mathfrak{b}_k^{\tau_+^j} \right) \\
 \\
 \frac{c_1 : (\Gamma, \vec{x}_1 : \vec{A}^1 \vdash \vec{\alpha}_1 : \vec{B}^1, \Delta) \quad \dots \quad c_l : (\Gamma, \vec{x}_l : \vec{A}^l \vdash \vec{\alpha}_l : \vec{B}^l, \Delta)}{\Gamma \mid \underline{\tilde{\mu} [\mathfrak{b}_1^{\tau_+^j}(\vec{x}_1, \vec{\alpha}_1).c_1 \mid \dots \mid \mathfrak{b}_l^{\tau_+^j}(\vec{x}_l, \vec{\alpha}_l).c_l] : \tau_+^j(\vec{C})} \vdash \Delta} \text{ (}\tau_+^j\text{)}
 \end{array}$$

Figure V.1.5.f: Logic rules for multiplicative types

$$\begin{array}{c}
 \frac{c : (\Gamma, x^+ : A_+ \vdash \alpha^- : B_-, \Delta)}{\Gamma \vdash \underline{\mu(x^+ \cdot \alpha^-).c : A_+ \rightarrow B_-} \mid \Delta} (\vdash \rightarrow) \quad \frac{\Gamma_1 \vdash \underline{v_+ : A_+} \mid \Delta_1 \quad \Gamma_2 \mid \underline{s_- : B_-} \vdash \Delta_2}{\Gamma_1, \Gamma_2 \mid \underline{v_+ \cdot s_- : A_+ \rightarrow B_-} \vdash \Delta_1, \Delta_2} (\rightarrow \vdash) \\
 \\
 \frac{c : (\Gamma \vdash \alpha^- : A_-, \beta^- : B_-, \Delta)}{\Gamma \vdash \underline{\mu(\alpha^- \wp \beta^-).c : A_- \& B_-} \mid \Delta} (\vdash \wp) \quad \frac{\Gamma_1 \mid \underline{s_-^1 : A_-^1} \vdash \Delta_1 \quad \Gamma_2 \mid \underline{s_-^2 : A_-^2} \vdash \Delta_2}{\Gamma_1, \Gamma_2 \mid \underline{(s_-^1 \wp s_-^2) : A_-^1 \& A_-^2} \vdash \Delta_1, \Delta_2} (\wp \vdash) \\
 \\
 \frac{\Gamma_1 \vdash \underline{v_+^1 : A_+^1} \mid \Delta_1 \quad \Gamma_2 \vdash \underline{v_+^2 : A_+^2} \mid \Delta_2}{\Gamma_1, \Gamma_2 \vdash \underline{(v_+^1 \otimes v_+^2) : A_+ \otimes B_+} \mid \Delta_1, \Delta_2} (\vdash \otimes) \quad \frac{c : (\Gamma, x^+ : A_+, y^+ : B_+ \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}(x^+ \otimes y^-).c : A_+ \otimes B_+} \vdash \Delta} (\otimes \vdash) \\
 \\
 \frac{c : (\Gamma \vdash \Delta)}{\Gamma \vdash \underline{\mu(\tilde{Q}).c : \perp} \mid \Delta} (\vdash \perp) \quad \frac{}{\mid \underline{\tilde{Q} : \perp} \vdash} (\perp \vdash) \\
 \\
 \frac{}{\vdash \underline{\tilde{Q} : 1} \mid} (1 \vdash) \quad \frac{c : (\Gamma \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}().c : 1} \vdash \Delta} (1 \vdash)
 \end{array}$$

Figure V.1.5.g: Logic rules for additive types

$$\begin{array}{c}
 \frac{c_1 : (\Gamma \vdash \alpha_1^- : A_-^1, \Delta) \quad c_2 : (\Gamma \vdash \alpha_2^- : A_-^2, \Delta)}{\Gamma \vdash \underline{\mu \langle (\pi_1 \cdot \alpha_1^-).c_1 \mid (\pi_2 \cdot \alpha_2^-).c_2 \rangle : A_-^1 \& A_-^2} \mid \Delta} (\vdash \&) \quad \frac{\Gamma \mid \underline{s_- : A_-^i} \vdash \Delta}{\Gamma \mid \underline{\pi_i \cdot s_- : A_-^1 \& A_-^2} \vdash \Delta} (\& \vdash) \\
 \\
 \frac{\Gamma \vdash \underline{v_+ : A_+^i} \mid \Delta}{\Gamma \vdash \underline{l_i(v_+) : A_+^1 \oplus A_+^2} \mid \Delta} (\vdash \oplus) \quad \frac{c_1 : (\Gamma, x_1^+ : A_+^1 \vdash \Delta) \quad c_2 : (\Gamma, x_2^+ : A_+^2 \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}[l_1(x_1^+).c_1 \mid l_2(x_2^+).c_2] : A_+^1 \oplus A_+^2} \vdash \Delta} (\oplus \vdash) \\
 \\
 \frac{}{\Gamma \vdash \underline{\mu \langle \rangle : \top} \mid \Delta} (\vdash \top) \quad \text{(No } (\top \vdash) \text{ rule)} \\
 \\
 \text{(No } (\vdash 0) \text{ rule)} \quad \frac{}{\Gamma \mid \underline{\tilde{\mu}[] : 0} \vdash \Delta} (0 \vdash)
 \end{array}$$

Figure V.1.5.h: Logic rules for shifts

$$\begin{array}{c}
 \frac{c : (\Gamma \vdash \alpha^+ : A_+, \Delta)}{\Gamma \vdash \underline{\mu\{\alpha^+\}.c : \uparrow A_+} \mid \Delta} (\uparrow\vdash) \qquad \frac{\Gamma \mid \underline{s_+ : A_+} \vdash \Delta}{\Gamma \mid \underline{\{s_+\} : \uparrow A_+} \vdash \Delta} (\uparrow\vdash) \\
 \\
 \frac{\Gamma \vdash \underline{v_- : A_-} \mid \Delta}{\Gamma \vdash \underline{\{v_-\} : \downarrow A_-} \mid \Delta} (\downarrow\vdash) \qquad \frac{c : (\Gamma, x^- : A_- \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}\{x^-\}.c : \downarrow A_-} \vdash \Delta} (\downarrow\vdash)
 \end{array}$$

Figure V.1.5.i: Logic rules for negations

$$\begin{array}{c}
 \frac{c : (\Gamma, x^+ : A_+ \vdash \Delta)}{\Gamma \vdash \underline{\mu_{\neg_-}(x^+).xc : \neg_-(A_+)} \mid \Delta} (\vdash\neg_-) \qquad \frac{\Gamma \vdash \underline{v_+ : A_+} \mid \Delta}{\Gamma \mid \underline{\neg_-(v_+) : \neg_-(A_+)} \vdash \Delta} (\neg_-\vdash) \\
 \\
 \frac{\Gamma \mid \underline{s_- : A_-} \vdash \Delta}{\Gamma \vdash \underline{\neg_+(s_-) : \neg_+(A_-)} \mid \Delta} (\vdash\neg_+) \qquad \frac{c : (\Gamma \vdash \alpha^- : A_-, \Delta)}{\Gamma \mid \underline{\tilde{\mu}_{\neg_+}(\alpha^-).xc : \neg_+(A_-)} \vdash \Delta} (\neg_+\vdash)
 \end{array}$$

Definition V.1.16

An expression t_ε (resp. evaluation context e_ε) of L_p^τ is said to be *of type* A_ε when there exists a derivation of

$$\Gamma \vdash t_\varepsilon : A_\varepsilon \mid \Delta \quad (\text{resp. } \Gamma \mid e_\varepsilon : A_\varepsilon \vdash \Delta)$$

in the type sytem described in Figure V.1.5, and *well-typed* when it is of type A_ε for some type A_ε . A command c is said to be *well-typed* when there exists a derivation of

$$c : (\Gamma \vdash \Delta)$$

in the type sytem described in Figure V.1.5. A term is said to be *ill-typed* when it is not well-typed.

Alternative presentations

Presentations of simply typed L-calculi sometimes use a syntax of preterms (i.e. possibly ill-typed terms) that treats polarities less rigidly, e.g. the syntax of preterms of [CurFioMun16, Figure 1, p. 4] allows pairs $(V_{\varepsilon_1} \otimes W_{\varepsilon_2})$ for values V_{ε_1} and W_{ε_2} of arbitrary polarities. Of course, extending the syntax of preterms without really changing the type system leaves the set of well-typed terms unchanged, e.g. extending simply typed L_p^τ with these pairs and the rules

$$\frac{\Gamma_1 \vdash V_{\varepsilon_1} : A_+ \mid \Delta_1 \quad \Gamma_2 \vdash W_{\varepsilon_2} : B_+ \mid \Delta_2}{\Gamma_1, \Gamma_2 \vdash (V_{\varepsilon_1} \otimes W_{\varepsilon_2}) : A_+ \otimes B_+} \quad \text{and} \quad \frac{c : (\Gamma, x^{\varepsilon_1} : A_+, y^{\varepsilon_2} : B_+ \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x^{\varepsilon_1} \otimes y^{\varepsilon_2}).c : A_+ \otimes B_+ \vdash \Delta}$$

would not change anything since these rules can only be used with $\varepsilon_1 = \varepsilon_2 = +$. However, often, the typing rules also allow types of arbitrary polarities, e.g. the type system of [CurFioMun16, Figure 2, p. 5] has the rules

$$\frac{\Gamma_1 \vdash V_{\varepsilon_1} : A_{\varepsilon_1} \mid \Delta_1 \quad \Gamma_2 \vdash W_{\varepsilon_2} : B_{\varepsilon_2} \mid \Delta_2}{\Gamma_1, \Gamma_2 \vdash (V_{\varepsilon_1} \otimes W_{\varepsilon_2}) : A_{\varepsilon_1} \otimes B_{\varepsilon_2}} \quad \text{and} \quad \frac{c : (\Gamma, x^{\varepsilon_1} : A_{\varepsilon_1}, y^{\varepsilon_2} : B_{\varepsilon_2} \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x^{\varepsilon_1} \otimes y^{\varepsilon_2}).c : A_{\varepsilon_1} \otimes B_{\varepsilon_2} \vdash \Delta}$$

This corresponds to an instance of L_p^τ with several instances of the type former indexed by the polarities of its argument, e.g. four type formers \otimes indexed by the polarities ε_1 and ε_2 of the left and right arguments. These can often be thought of as being combinations of a single type former with shifts, e.g. we have isomorphisms

$$A_- \otimes^{\neg,+} B_+ \cong \downarrow A_- \otimes^{+,+} B_+, \quad A_+ \otimes^{+,-} B_- \cong A_+ \otimes^{+,+} \downarrow B_+ \quad \text{and} \quad A_- \otimes^{--} B_- \cong \downarrow A_- \otimes^{+,+} \downarrow B_+$$

Conversely, while these presentations often do not define the shifts, they are often expressible:

$$\uparrow A_+ \cong 1 \rightarrow A_+ \quad \text{and} \quad \downarrow A_- \cong 1 \otimes A_-$$

Most presentations therefore end up being more or less equivalent.

[CurFioMun16] “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”, Curien, Fiore, and Munch-Maccagnoni, 2016

Well-polarized terms

The type system in Figure V.1.5 can be weakened by replacing each type A_ϵ by its polarity ϵ , which yields the type system described in Figure .4.1 of the appendix. Terms that are well-typed in this weaker system are called well-polarized:

Definition V.1.17

A term of $L_p^{\vec{\tau}}$ is said to be *well-polarized* (resp. *ill-polarized*) when it is well-typed (resp. ill-typed) in the type system described in Figure .4.1.

Fact V.1.18

Well-typed terms are well-polarized.

Proof

By induction on the derivation, replacing each type A_ϵ by its polarity ϵ .

Many presentations of L-calculi in the litterature mostly focus on well-typed terms, and can hence choose a presentation that allows ill-polarized terms in the syntax for the sake of simplicity. Here, however, we want to study an untyped L-calculus, and must therefore reject the ill-polarized terms explicitly. The $L_p^{\vec{\tau}}$ calculus (or rather its instances for specific choices of $\vec{\tau}$) can be obtained from L-calculi of the litterature by restricting to well-polarized terms.

The rigid treatment of polarities in the syntax of $L_p^{\vec{\tau}}$ ensures that all terms are well-polarized:

Fact V.1.19

For any command c (resp. expression t_ϵ , evaluation context e_ϵ) of $L_p^{\vec{\tau}}$, we have

$$c : (\Gamma \vdash \Delta) \quad (\text{resp. } \Gamma \vdash t_\epsilon : \epsilon \mid \Delta, \quad \Gamma \mid e_\epsilon : \epsilon \vdash \Delta)$$

if and only if Γ and Δ map all free variables of c (resp. t_ϵ, e_ϵ) to their polarities, i.e.

$$\Gamma = \vec{x}^+ : +, \vec{y}^- : - \quad \text{and} \quad \Delta = \vec{\alpha}^+ : +, \vec{\beta}^- : -$$

with

$$\text{FV}(c) \subseteq \{\vec{x}^+, \vec{y}^+, \vec{\alpha}^+, \vec{\beta}^-\} \quad (\text{resp. } \text{FV}(t_\epsilon) \subseteq \{\vec{x}^+, \vec{y}^+, \vec{\alpha}^+, \vec{\beta}^-\}, \quad \text{FV}(e_\epsilon) \subseteq \{\vec{x}^+, \vec{y}^+, \vec{\alpha}^+, \vec{\beta}^-\})$$

In particular, all terms of $L_p^{\vec{\tau}}$ are well-polarized.

Proof

The \Rightarrow implication is by induction on the typing derivation, and the \Leftarrow implication it by induction on the syntax (using the structural rules to preserve the whole context

through multiplicative rules).

V.2. Intuitionistic and minimalistic polarized L-calculi: $\text{Li}_p^{\vec{\tau}}$ and $\text{Lm}_p^{\vec{\tau}}$


In this section, we start the process of transforming $\text{L}_p^{\vec{\tau}}$ into a corresponding λ -calculus ($\lambda_p^{\vec{\tau}}$ of Section V.5), by carving out its intuitionistic fragment $\text{Li}_p^{\vec{\tau}}$ and its minimalistic fragment $\text{Lm}_p^{\vec{\tau}}$. In a typed setting, those fragments correspond to the restrictions of classical logic to minimal and intuitionistic logic respectively.

V.2.1. Intuitionistic and minimalistic fragments

Fragment definitions

It is well-known Gentzen's sequent calculus for classical logic can be restricted to minimal logic (resp. intuitionistic logic) by only considering sequents with exactly one (resp. at most one) succedent. We define the minimalistic⁴ (resp. intuitionistic) fragment of $\text{L}_p^{\vec{\tau}}$ similarly:

Definition V.2.1

Given a set of nice type formers $\vec{\tau}$, a term of $\text{L}_p^{\vec{\tau}}$ is said to be *minimalistically well-typed* (resp. *intuitionistically well-typed*) when there is a derivation of its well-typedness that only contains sequents with exactly one (resp. at most one) succedent^a. This yields the type system described in  for minimalistically well-typed terms.

^aThe number of conclusions of a sequent is the number of types on the right of the \vdash symbol, so that sequents with one succedent are those of the shape

$$c : (\Gamma \vdash \alpha^\varepsilon : A_\varepsilon), \quad \Gamma \vdash t_\varepsilon : A_\varepsilon, \quad \text{or} \quad \Gamma \mid \underline{e_{\varepsilon_1} : A_{\varepsilon_1}} \vdash \alpha^{\varepsilon_2} : A_{\varepsilon_2}$$

and sequents with zero succedents are those of the shape

$$c : (\Gamma \vdash) \quad \text{or} \quad \Gamma \mid \underline{e_\varepsilon : A_\varepsilon} \vdash$$

These restrictions can also be applied to the trivial type system that only accounts for polarities:

Definition V.2.2

A term of $\text{L}_p^{\vec{\tau}}$ is said to be *minimalistically well-polarized* (resp. *intuitionistically well-polarized*), or *minimalistic* (resp. *intuitionistic*), when there is a derivation of its well-polarization that only contains sequents with exactly one (resp. at most one) succedent. This yields the type system described in Figure V.2.1 for minimalistically well-typed terms. We call *minimalistic fragment* (resp. *intuitionistic fragment*) of $\text{L}_p^{\vec{\tau}}$, and denote by $\text{Lm}_p^{\vec{\tau}}$ (resp. $\text{Li}_p^{\vec{\tau}}$), the subset of $\text{L}_p^{\vec{\tau}}$ that consists of all minimalistically (resp. intuitionistically) well-polarized terms.

⁴Since we use this adjective for many kinds of objects, including some that are equipped with a preorder (e.g. terms with the observational preorder), we use “minimalistic” instead of “minimal” to avoid any ambiguity.

Figure V.2.1: Well polarized $\text{Lm}_p^{\bar{\tau}}$

Figure V.2.1.a: Core rules

$$\begin{array}{c}
 \frac{}{x^\varepsilon : \varepsilon \vdash \underline{x^\varepsilon : \varepsilon} \mid} \text{(\vdash AX)} \qquad \frac{}{\mid \underline{\alpha^\varepsilon : \varepsilon} \vdash \alpha^\varepsilon : \varepsilon} \text{(AX\vdash)} \\
 \\
 \frac{c : (\Gamma \vdash \alpha^\varepsilon : \varepsilon)}{\Gamma \vdash \underline{\mu \alpha^\varepsilon . c : \varepsilon} \mid} \text{(\vdash } \mu \text{)} \qquad \frac{c : (\Gamma, x^\varepsilon : \varepsilon \vdash \alpha^{\varepsilon_r} : \varepsilon_r)}{\Gamma \mid \underline{\tilde{\mu} x^\varepsilon . c : \varepsilon} \vdash \alpha^{\varepsilon_r} : \varepsilon_r} \text{(\tilde{\mu}\vdash)} \\
 \\
 \frac{\Gamma_1 \vdash \underline{t_\varepsilon : \varepsilon} \mid \quad \Gamma_2 \mid \underline{e_\varepsilon : \varepsilon} \vdash \alpha^{\varepsilon_r} : \varepsilon_r}{\langle \underline{t_\varepsilon} \mid \underline{e_\varepsilon} \rangle^\varepsilon : (\Gamma_1, \Gamma_2 \vdash \alpha^{\varepsilon_r} : \varepsilon_r)} \text{(CUT)}
 \end{array}$$

Figure V.2.1.b: Structural rules (commands)

$$\begin{array}{c}
 \text{(Inoperable (\vdash wc) rule)} \qquad \text{(Inoperable (\vdash cc) rule)} \\
 \\
 \frac{c : (\Gamma \vdash \alpha^{\varepsilon_r} : \varepsilon_r)}{c : (\Gamma, x^\varepsilon : \varepsilon \vdash \alpha^{\varepsilon_r} : \varepsilon_r)} \text{(wc\vdash)} \qquad \frac{c : (\Gamma, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon \vdash \alpha^{\varepsilon_r} : \varepsilon_r)}{c[y^\varepsilon/x_1^\varepsilon, y^\varepsilon/x_2^\varepsilon] : (\Gamma, y^\varepsilon : \varepsilon \vdash \alpha^{\varepsilon_r} : \varepsilon_r)} \text{(cc\vdash)} \\
 \\
 \text{(Inoperable (\vdash pc) rule)} \qquad \frac{c : (\Gamma_1, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon, \Gamma_2 \vdash \alpha^{\varepsilon_r} : \varepsilon_r)}{c : (\Gamma_1, x_2^\varepsilon : \varepsilon, x_1^\varepsilon : \varepsilon, \Gamma_2 \vdash \alpha^{\varepsilon_r} : \varepsilon_r)} \text{(pc\vdash)}
 \end{array}$$

Figure V.2.1.c: Structural rules (expressions)

$$\begin{array}{c}
 \text{(Inoperable (\vdash wt) rule)} \qquad \text{(Inoperable (\vdash ct) rule)} \\
 \\
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid}{\Gamma, x^\varepsilon : \varepsilon \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid} \text{(wt\vdash)} \qquad \frac{\Gamma, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid}{\Gamma, x^\varepsilon : \varepsilon \vdash \underline{t_{\varepsilon_0} [x^\varepsilon/x_1^\varepsilon, x^\varepsilon/x_2^\varepsilon] : \varepsilon_0} \mid} \text{(ct\vdash)} \\
 \\
 \text{(Inoperable (\vdash pt) rule)} \qquad \frac{\Gamma_1, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon, \Gamma_2 \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid}{\Gamma_1, x_2^\varepsilon : \varepsilon, x_1^\varepsilon : \varepsilon, \Gamma_2 \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid} \text{(pt\vdash)}
 \end{array}$$

Figure V.2.1.d: Structural rules (evaluation contexts)

<p>(Inoperable (\vdash_{we}) rule)</p> $\frac{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \alpha^{\varepsilon_r} : \varepsilon_r}{\Gamma, x^\varepsilon : \varepsilon \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \alpha^{\varepsilon_r} : \varepsilon_r} \text{ (we}\vdash\text{)}$	<p>(Inoperable (\vdash_{ce}) rule)</p> $\frac{\Gamma, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \alpha^{\varepsilon_r} : \varepsilon_r}{\Gamma, x^\varepsilon : \varepsilon \mid \underline{e_{\varepsilon_0} [x^\varepsilon / x_1^\varepsilon, x^\varepsilon / x_2^\varepsilon] : \varepsilon_0} \vdash \alpha^{\varepsilon_r} : \varepsilon_r} \text{ (ce}\vdash\text{)}$
<p>(Inoperable (\vdash_{pe}) rule)</p>	$\frac{\Gamma_1, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon, \Gamma_2 \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta}{\Gamma_1, x_2^\varepsilon : \varepsilon, x_1^\varepsilon : \varepsilon, \Gamma_2 \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta} \text{ (pe}\vdash\text{)}$

Figure V.2.1.e: General shape of logic rules (assuming vs-sorted constructors)

$$\begin{array}{c}
 \frac{\Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : \varepsilon_1} \mid \dots \mid \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : \varepsilon_q} \mid \Gamma_{q+1} \mid \underline{s_{\varepsilon_{q+1}}^1 : \varepsilon_{q+1}} \vdash \alpha^{\varepsilon_r} : \varepsilon_r}{\Gamma_1, \dots, \Gamma_{q+r} \mid \underline{\mathfrak{s}_k^{\tau_-^j} (v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1)} : - \vdash \alpha^{\varepsilon_r} : \varepsilon_r} \left(\mathfrak{s}_k^{\tau_-^j} \vdash \right) \\
 \\
 \frac{c_1 : (\Gamma, \bar{x}_1^\rightarrow : \bar{\varepsilon}_1^\rightarrow \vdash \alpha_1^{\varepsilon_r} : \varepsilon_r) \quad \dots \quad c_l : (\Gamma, \bar{x}_l^\rightarrow : \bar{\varepsilon}_l^\rightarrow \vdash \alpha_l^{\varepsilon_r} : \varepsilon_r)}{\Gamma \vdash \underline{\mu \langle \mathfrak{s}_1^{\tau_-^j} (\bar{x}_1^\rightarrow, \alpha_1). c_1 \mid \dots \mid \mathfrak{s}_l^{\tau_-^j} (\bar{x}_l^\rightarrow, \alpha_l). c_l \rangle} : - \mid} \left(\vdash_{\tau_-^j} \right) \\
 \\
 \frac{\Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : \varepsilon_1} \mid \dots \mid \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : \varepsilon_q} \mid}{\Gamma_1, \dots, \Gamma_q \vdash \underline{\mathfrak{b}_k^{\tau_+^j} (v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1, \dots, s_{\varepsilon_{q+r}}^r)} : + \mid} \left(\vdash_{\mathfrak{b}_k^{\tau_+^j}} \right) \\
 \\
 \frac{c_1 : (\Gamma, \bar{x}_1^\rightarrow : \bar{\varepsilon}_1^\rightarrow \vdash \alpha_1^{\varepsilon_r} : \varepsilon_r) \quad \dots \quad c_l : (\Gamma, \bar{x}_l^\rightarrow : \bar{\varepsilon}_l^\rightarrow \vdash \alpha_l^{\varepsilon_r} : \varepsilon_r)}{\Gamma \mid \underline{\tilde{\mu} [\mathfrak{b}_1^{\tau_+^j} (\bar{x}_1^\rightarrow, \bar{\alpha}_1). c_1 \mid \dots \mid \mathfrak{b}_l^{\tau_+^j} (\bar{x}_l^\rightarrow, \bar{\alpha}_l). c_l]} : + \vdash \alpha^{\varepsilon_r} : \varepsilon_r} \left(\tau_+^j \vdash \right)
 \end{array}$$

Figure V.2.1.f: Logic rules for multiplicative types

$$\begin{array}{cc}
 \frac{c:(\Gamma, x^+ : + \vdash \alpha^- : -)}{\Gamma \vdash \underline{\mu(x^+ \cdot \alpha^-).c} : -} (\vdash \rightarrow) & \frac{\Gamma_1 \vdash \underline{v_+} : + \mid \quad \Gamma_2 \mid \underline{s_-} : - \vdash \alpha^\varepsilon : \varepsilon}{\Gamma_1, \Gamma_2 \mid \underline{v_+ \cdot s_-} : - \vdash \alpha^\varepsilon : \varepsilon} (\rightarrow \vdash) \\
 \text{(Inoperable } (\vdash \rightarrow) \text{ rule)} & \text{(Inoperable } (\rightarrow \vdash) \text{ rule)} \\
 \\
 \frac{\Gamma_1 \vdash \underline{v_+^1} : + \mid \quad \Gamma_2 \vdash \underline{v_+^2} : + \mid}{\Gamma_1, \Gamma_2 \vdash \underline{(v_+^1 \otimes v_+^2)} : + \mid} (\vdash \otimes) & \frac{c:(\Gamma, \alpha^+ : +, y^+ : + \vdash \alpha^\varepsilon : \varepsilon)}{\Gamma \mid \underline{\tilde{\mu}(x^+ \otimes y^-).c} : + \vdash \alpha^\varepsilon : \varepsilon} (\otimes \vdash) \\
 \text{(Inoperable } (\vdash \perp) \text{ rule)} & \text{(Inoperable } (\perp \vdash) \text{ rule)} \\
 \\
 \frac{}{\vdash \underline{()} : + \mid} (1 \vdash) & \frac{c:(\Gamma \vdash \alpha^\varepsilon : \varepsilon)}{\Gamma \mid \underline{\tilde{\mu}().c} : + \vdash \alpha^\varepsilon : \varepsilon} (\vdash 1)
 \end{array}$$

Figure V.2.1.g: Logic rules for additive types

$$\begin{array}{cc}
 \frac{c_1:(\Gamma \vdash \alpha_1^- : -) \quad c_2:(\Gamma \vdash \alpha_2^- : -)}{\Gamma \vdash \underline{\mu \langle (\pi_1 \cdot \alpha_1^-).c_1 \mid (\pi_2 \cdot \alpha_2^-).c_2 \rangle} : -} (\vdash \&) & \frac{\Gamma \mid \underline{s_-} : - \vdash \alpha^\varepsilon : \varepsilon}{\Gamma \mid \underline{\pi_i \cdot s_-} : - \vdash \alpha^\varepsilon : \varepsilon} (\& \vdash) \\
 \\
 \frac{\Gamma \vdash \underline{v_+} : + \mid}{\Gamma \vdash \underline{\iota_i(v_+)} : + \mid} (\vdash \oplus) & \frac{c_1:(\Gamma, x_1^+ : + \vdash \alpha^\varepsilon : \varepsilon) \quad c_2:(\Gamma, x_2^+ : + \vdash \alpha^\varepsilon : \varepsilon)}{\Gamma \mid \underline{\tilde{\mu}[\iota_1(x_1^+).c_1 \mid \iota_2(x_2^+).c_2]} : + \vdash \alpha^\varepsilon : \varepsilon} (\oplus \vdash) \\
 \\
 \frac{}{\Gamma \vdash \underline{\mu \langle \rangle} : - \mid} (\vdash \top) & \text{(No } (\top \vdash) \text{ rule)} \\
 \text{(No } (\vdash 0) \text{ rule)} & \frac{}{\Gamma \mid \underline{\tilde{\mu}[]} : + \vdash \alpha^\varepsilon : \varepsilon} (0 \vdash)
 \end{array}$$

Figure V.2.1.h: Logic rules for shifts

$$\begin{array}{cc}
 \frac{c:(\Gamma \vdash \alpha^+ : +)}{\Gamma \vdash \underline{\mu\{\alpha^+\}.c} : -} (\vdash \Uparrow) & \frac{\Gamma \mid \underline{s_+} : + \vdash \alpha^\varepsilon : \varepsilon}{\Gamma \mid \underline{\{s_+\}} : - \vdash \alpha^\varepsilon : \varepsilon} (\Uparrow \vdash) \\
 \\
 \frac{\Gamma \vdash \underline{v_-} : - \mid}{\Gamma \vdash \underline{\{v_-\}} : + \mid} (\vdash \Downarrow) & \frac{c:(\Gamma, x^- : - \vdash \alpha^\varepsilon : \varepsilon)}{\Gamma \mid \underline{\tilde{\mu}\{x^-\}.c} : + \vdash \alpha^\varepsilon : \varepsilon} (\Downarrow \vdash)
 \end{array}$$

Figure V.2.1.i: Logic rules for negations	
(Inoperable ($\vdash \neg_-$) rule)	(Inoperable ($\neg_- \vdash$) rule)
(Inoperable ($\vdash \neg_+$) rule)	(Inoperable ($\neg_+ \vdash$) rule)

Inoperable rules

The restriction prevent the use of some some rules:

Definition V.2.3

A typing rule of $L_p^{\bar{\tau}}$ is said to be *operable in $Lm_p^{\bar{\tau}}$* (resp. *operable in $Li_p^{\bar{\tau}}$*) when there exists a derivation that a term is minimalistically (resp. intuitionistically) well-polarized that uses an instance of that typing rule.

Example V.2.4

The core rules, the left structural rules, and the logic rules for $\rightarrow, \Downarrow, \Uparrow, \otimes, \oplus, \&, 1, 0,$ and \top are operable in $Lm_p^{\bar{\tau}}$, while the right structural rules and the logic rules for $\neg_{-}, \neg_{+}, \wp,$ and \perp are not.

Note that removing type formers whose typing rules are not operable does not change the calculus, e.g.

$$Lm_p^{\rightarrow \Downarrow \Uparrow \neg_{-} \neg_{+} \otimes \wp \oplus \& 1 \perp 0 \top} = Lm_p^{\rightarrow \Downarrow \Uparrow \otimes \oplus \& 1 0 \top}$$

Operability of logic rules in $Lm_p^{\bar{\tau}}$ can be fully characterized via fairly simple criteria:

Fact V.2.5

For logic rules, we have:

$$\begin{aligned} (\mathfrak{g}_k^{\tau_{-}^j} \vdash) \text{ is operable in } Lm_p^{\bar{\tau}} &\Leftrightarrow \mathfrak{g}_k^{\tau_{-}^j} \text{ has a single stack argument} \\ (\vdash \tau_{-}^j) \text{ is operable in } Lm_p^{\bar{\tau}} &\Leftrightarrow \forall k, \mathfrak{g}_k^{\tau_{-}^j} \text{ has a single stack argument} \\ (\vdash \mathfrak{b}_k^{\tau_{+}^j}) \text{ is operable in } Lm_p^{\bar{\tau}} &\Leftrightarrow \mathfrak{b}_k^{\tau_{+}^j} \text{ has a no stack argument} \\ (\tau_{+}^j \vdash) \text{ is operable in } Lm_p^{\bar{\tau}} &\Leftrightarrow \forall k, \mathfrak{b}_k^{\tau_{+}^j} \text{ has a no stack argument} \end{aligned}$$

Proof

Proofs that rules are not operable and the \Rightarrow implications are by case analysis on the number of succedents in each sequent of the rule. Proofs that rules are operable and the \Leftarrow implications simply exhibit a derivation in $Lm_p^{\bar{\tau}}$ that uses the rule. For $(\vdash \tau_{-}^j)$ and $(\vdash \mathfrak{b}_k^{\tau_{+}^j})$, any derivation that

$$\mathfrak{g}_k^{\tau_{-}^j}(\vec{x}, \alpha^{\varepsilon}, \vec{y}) \quad \text{and} \quad \mathfrak{b}_k^{\tau_{+}^j}(\vec{x})$$

are minimalistically well-polarized works, and for $(\tau_{-}^j \vdash)$ and $(\tau_{+}^j \vdash)$, any derivation

that

$$\mu \left\langle \begin{array}{c} \mathfrak{s}_1^{\tau_1^j}(\vec{x}_1, \alpha_1^{\varepsilon_1}, \vec{y}_1). \langle z_1^{\varepsilon_1} | \alpha_1^{\varepsilon_1} \rangle^{\varepsilon_1} \\ \vdots \\ \mathfrak{s}_l^{\tau_l^j}(\vec{x}_l, \alpha_l^{\varepsilon_l}, \vec{y}_l). \langle z_l^{\varepsilon_l} | \alpha_l^{\varepsilon_l} \rangle^{\varepsilon_l} \end{array} \right\rangle \quad \text{and} \quad \tilde{\mu} \left[\begin{array}{c} \mathfrak{b}_1^{\tau_1^j}(\vec{x}_1). \langle y_1^{\varepsilon_1} | \alpha^{\varepsilon_1} \rangle^{\varepsilon_1} \\ \vdots \\ \mathfrak{b}_l^{\tau_l^j}(\vec{x}_l). \langle y_l^{\varepsilon_l} | \alpha^{\varepsilon_l} \rangle^{\varepsilon_l} \end{array} \right]$$

are minimalistically well-polarized works.

Operability of rules in $\text{Li}_p^{\bar{\tau}}$ is a bit more complex:

Example V.2.6

All rules that were operable in $\text{Lm}_p^{\bar{\tau}}$ are still operable in $\text{Li}_p^{\bar{\tau}}$. Among rules that were inoperable in $\text{Lm}_p^{\bar{\tau}}$, the right contraction rules, the right permutation rules, the weakening rule for terms ($\vdash \text{wt}$), and the logic rule ($\vdash \mathfrak{A}$) remain inoperable in $\text{Li}_p^{\bar{\tau}}$, while the left logic rules ($\perp \vdash$), ($\neg \vdash$), and ($\neg_+ \vdash$) become operable in $\text{Li}_p^{\bar{\tau}}$. The operability of the remaining rules depends on the ability to instantiate enough of the premises with sequents that have no succedents. The rule ($\vdash \perp$) (resp. ($\vdash \neg$)) is always operable in $\text{Li}_p^{\bar{\tau}}$ because this ability is provided by the corresponding left logic rule ($\perp \vdash$) (resp. ($\neg \vdash$))^a. The rules ($\vdash \text{wc}$), ($\vdash \text{we}$), ($\mathfrak{A} \vdash$), and ($\vdash \neg_+$) may be operable in $\text{Li}_p^{\bar{\tau}}$ or not depending on what type formers are in $\bar{\tau}$: none are operable in $\text{Li}_p^{\mathfrak{A}\neg_+}$, the first two are operable in $\text{Li}_p^{\mathfrak{A}\neg_+0}$ while the latter two are not, and all four are operable in $\text{Li}_p^{\mathfrak{A}\neg_+\perp}$ and in $\text{Li}_p^{\uparrow\mathfrak{A}\neg_+0}$.

^aIndeed, the η -expansion of x^-

$\mu\tilde{\mathfrak{O}}.\langle x^- | \tilde{\mathfrak{O}} \rangle^-$, (resp. $\mu\neg_-(y^+).y\langle x^- | \neg_-(y^+) \rangle^-$)

is always in $\text{Li}_p^{\bar{\tau}}$ (assuming that $\perp \in \bar{\tau}$ (resp. $\neg_- \in \bar{\tau}$)).

This characterization can most likely be generalized to arbitrary type formers by defining

$$\mathbf{X} = \{\varepsilon \in \{+, -\} \mid \text{the judgement } \Gamma \mid \underline{s_\varepsilon} : \varepsilon \vdash \text{ is derivable in } \text{Li}_p^{\bar{\tau}} \text{ for some } s_\varepsilon \text{ and } \Gamma\}$$

(which is such that $- \in \mathbf{X}$ implies $+$ $\in \mathbf{X}$ because we can form $\tilde{\mu}x^+.\langle y^- | s_- \rangle^-$) and using conditions such as “ $\mathfrak{s}_k^{\tau_k^j}$ has at most one stack argument whose polarity is not in \mathbf{X} ”, but we have no use for such a characterization, and therefore do not work out the details here.

Inclusions

We of course have inclusions:

Fact V.2.7

For any $\bar{\tau}$, we have

$$\text{Lm}_p^{\bar{\tau}} \subseteq \text{Li}_p^{\bar{\tau}} \subsetneq \text{L}_p^{\bar{\tau}}$$

Proof

The inclusions are immediate. We have $\text{Li}_p^{\bar{\tau}} \not\subseteq \text{L}_p^{\bar{\tau}}$ because for $\alpha^\varepsilon \neq \beta^\varepsilon$,

$$\text{Li}_p^{\bar{\tau}} \not\subseteq \langle \mu \alpha^\varepsilon. \langle x^\varepsilon | \beta^\varepsilon \rangle^\varepsilon | \beta^\varepsilon \rangle^\varepsilon \in \text{L}_p^{\bar{\tau}}$$

The first inclusion may be an equality or not depending on $\bar{\tau}$:

Fact V.2.8

The following are equivalent:

- (i) there exists a derivation of well-polarization which is valid in $\text{Li}_p^{\bar{\tau}}$ but not in $\text{Lm}_p^{\bar{\tau}}$;
- (ii) there exists a derivation of well-polarization which is valid in $\text{Li}_p^{\bar{\tau}}$ but not in $\text{Lm}_p^{\bar{\tau}}$, and whose conclusion is of the shape

$$c : (\Gamma \vdash) \quad \text{or} \quad \Gamma \mid \underline{e_\varepsilon : \varepsilon} \vdash$$

i.e. has no succedent;

- (iii) there exists a stack s_ε in $\text{Li}_p^{\bar{\tau}}$ such that $\Gamma \mid \underline{s_\varepsilon : \varepsilon} \vdash$ is derivable for some Γ ;
- (iv) at least one of the following holds:
 - (a) there exists a stack constructor $\mathfrak{s}_k^{\tau_j^j}$ with zero stack arguments (e.g. $\neg_-(v_+)$ or $\tilde{()}$); or
 - (b) there exists a positive type former τ_+^j whose value constructors $\mathfrak{v}_k^{\tau_+^j}$ all have exactly one stack arguments (e.g. \neg_+ or 0).
- (v) there exists a stack s_ε in $\text{Li}_p^{\bar{\tau}}$ of the shape

$$s_\varepsilon = \mathfrak{s}_k^{\tau_j^j}(\vec{x}) \quad (\text{e.g. } \neg_-(x^+) \quad \text{or} \quad \tilde{()})$$

or

$$s_\varepsilon = \tilde{\mu} \left[\begin{array}{c} \mathfrak{v}_1^{\tau_+^j}(\vec{x}_1, \alpha_1^{\varepsilon_1}, \vec{y}_1). \langle z_1^{\varepsilon_1} | \alpha_1^{\varepsilon_1} \rangle^{\varepsilon_1} \\ \vdots \\ \mathfrak{v}_l^{\tau_+^j}(\vec{x}_l, \alpha_l^{\varepsilon_l}, \vec{y}_l). \langle z_l^{\varepsilon_l} | \alpha_l^{\varepsilon_l} \rangle^{\varepsilon_l} \end{array} \right] \quad (\text{e.g. } \tilde{\mu}_{\neg_+}(\alpha^-). \alpha \langle x^- | \alpha^- \rangle^- \quad \text{or} \quad \tilde{\mu}[])$$

Furthermore, if all positive type formers in $\bar{\tau}$ have at least one constructor (i.e. there are no copies of 0), then these are also equivalent to:

- (vi) $\text{Lm}_p^{\bar{\tau}} \subsetneq \text{Li}_p^{\bar{\tau}}$.

In particular, for $\bar{\tau} \subseteq \{\rightarrow, \Downarrow, \Uparrow, \neg_-, \neg_+, \otimes, \wp, \oplus, \&, 1, \perp, \top\}^a$, we have

$$\text{Lm}_p^{\bar{\tau}} \subsetneq \text{Li}_p^{\bar{\tau}} \Leftrightarrow \bar{\tau} \cap \{\neg_-, \neg_+, \perp\} \neq \emptyset$$

^aNote the absence of 0 .

Proof sketch (See page 232 for details)

The implications (i) \Rightarrow (ii) \Rightarrow (iii) \Rightarrow (iv) \Rightarrow (v) \Rightarrow (i) \Leftarrow (iv) are either immediate or by induction on the derivation, and in the particular case, the implication (v) \Rightarrow (iv) is immediate.

Straightforwardly minimalistic type formers

The restriction to sequents with exactly one (resp. at most one) conclusion is mostly used in systems in which “classical” type formers have already been removed, and for an arbitrary set of type formers $\bar{\tau}$, there might be some subtleties that Definition V.2.2 fails to consider⁵. However, for some sets of type formers, no such subtleties arise:

Definition V.2.9

A negative type former is said to be *straightforwardly minimalistic* when all its rules are operable in $\text{Lm}_{\bar{\tau}}^{\bar{\tau}}$, and a positive type former is said to be *straightforwardly minimalistic* when all its rules are operable in $\text{Lm}_{\bar{\tau}}^{\bar{\tau}}$ and it has at least one constructor (i.e. it is not a copy of $\mathbf{0}$).

Example V.2.10

The type formers $\rightarrow, \Downarrow, \Uparrow, \otimes, \oplus, \&, \mathbf{1}$, and \top are straightforwardly minimalistic, while $\neg, \neg_+, \wp, \mathbf{0}$, and \perp are not.

The restriction to type formers that are operable in $\text{Lm}_{\bar{\tau}}^{\bar{\tau}}$ is fairly natural: we remove unwanted type formers before applying the restriction. The rejection of $\mathbf{0}$ is a bit harder to justify, but is not completely unheard of⁶, not completely arbitrary⁷, and fairly harmless:

⁵For example, negations can not be used in $\text{Lm}_{\bar{\tau}}^{\bar{\tau}}$ while some sequents with negations are provable in minimal logic according to the [ncatlab page on minimal logic](#).

⁶The type former $\mathbf{0}$ needs to be removed to ensure that the teleological version of **ILL** is faithful [Gir11, p. 217].

⁷One could restrict

$$\frac{}{\Gamma \mid \tilde{\mu}[\] : \mathbf{0} \vdash \Delta} (\mathbf{0}\vdash) \quad \text{to} \quad \frac{}{\mid \tilde{\mu}[\] : \mathbf{0} \vdash}$$

This would have no effect in $\text{L}_{\bar{\tau}}^{\bar{\tau}}$ since the full rule would be derivable by composing the restricted rule with weakening rules, but this would make the rule $(\mathbf{0}\vdash)$ inoperable in $\text{Lm}_{\bar{\tau}}^{\bar{\tau}}$.

Furthermore, this restriction can be seen as an instance of a natural and systematic transformation of additive rules that allows them to have different contexts in their premises and takes their unions in the conclusions, e.g. replacing

$$\frac{c_1 : (\Gamma, x_1^+ : A_+^1 \vdash \Delta) \quad c_2 : (\Gamma, x_2^+ : A_+^2 \vdash \Delta)}{\Gamma \mid \tilde{\mu}[l_1(x_1^+).c_1 \mid l_2(x_2^+).c_2] : A_+^1 \oplus A_+^2 \vdash \Delta} (\oplus\vdash) \quad \text{by} \quad \frac{c_1 : (\Gamma_1, x_1^+ : A_+^1 \vdash \Delta_1) \quad c_2 : (\Gamma_2, x_2^+ : A_+^2 \vdash \Delta_2)}{\Gamma_1 \cup \Gamma_2 \mid \tilde{\mu}[l_1(x_1^+).c_1 \mid l_2(x_2^+).c_2] : A_+^1 \oplus A_+^2 \vdash \Delta_1 \cup \Delta_2}$$

For other additive types, this yields a more general rule which is derivable in $\text{L}_{\bar{\tau}}^{\bar{\tau}}$ by composing the normal rules with weakening rules, but for $(\mathbf{0}\vdash)$, since there are no premises, we get the neutral element for context union, i.e. the empty context.

V. Polarized calculi with arbitrary constructors

Fact V.2.11

Given a set of straightforwardly minimalistic type formers $\vec{\tau}$, for any term t , we have

$$t' \in \text{Lm}_p^{\vec{\tau}_0} \Leftrightarrow \exists t \in \text{Lm}_p^{\vec{\tau}}, t \xrightarrow{0}_0^* t'$$

i.e. terms of $\text{Lm}_p^{\vec{\tau}_0}$ are exactly those of $\text{Lm}_p^{\vec{\tau}}$ with some positive stacks replaced by $\tilde{\mu}[]$.

Proof

The \Rightarrow implication is by induction on the derivation that $t' \in \text{Lm}_p^{\vec{\tau}_0}$, and the \Leftarrow implication follows from $\text{Lm}_p^{\vec{\tau}} \subseteq \text{Lm}_p^{\vec{\tau}_0}$ and closure of $\text{Lm}_p^{\vec{\tau}_0}$ under $\xrightarrow{0}_0^*$, which is proven by induction on the derivation of $t \xrightarrow{0}_0^* t'$.

Restricting to straightforwardly minimalistic type formers forces all commands and evaluation contexts to have at least one free stack variable (which is crucial for $\Delta\Delta\Delta$):

Fact V.2.12

Given a set of straightforwardly minimalistic type formers $\vec{\tau}$, for any evaluation context e_ε (resp. command c) of $\text{L}_p^{\vec{\tau}}$, we have

$$|\text{FV}_s(e_\varepsilon)| \geq 1 \quad (\text{resp. } |\text{FV}_s(c)| \geq 1)$$

In particular, there are no derivations whose conclusion is of the shape

$$\Gamma \mid \underline{e_\varepsilon} : \varepsilon \vdash \quad (\text{resp. } c : (\Gamma \vdash))$$

Proof

By induction on the derivation that e_ε (resp. c) is well-polarized. The restriction on derivations follows by Fact V.1.19.

Note that this property fails if $0 \in \vec{\tau}$:

$$\Gamma \mid \underline{\tilde{\mu}[]} : + \vdash \alpha^\varepsilon : \varepsilon \quad \text{but} \quad \text{FV}_s(\tilde{\mu}[]) = \emptyset$$

By forbidding these judgements, the restriction to straightforwardly minimalistic type formers erases the distinction between $\text{Lm}_p^{\vec{\tau}}$ and $\text{Li}_p^{\vec{\tau}}$:

Fact V.2.13

For any set of straightforwardly minimalistic type formers $\vec{\tau}$, we have $\text{Lm}_p^{\vec{\tau}} = \text{Li}_p^{\vec{\tau}}$.

Proof

By the previous fact.

V.2.2. A syntax for the minimalistic fragment

Characterization of $\text{Lm}_p^{\bar{\tau}}$ via free stack variables

The $\text{Lm}_p^{\bar{\tau}}$ calculus can be characterized as a subcalculus of $\text{L}_p^{\bar{\tau}}$ as follows:

Proposition V.2.14

Given a set of straightforwardly minimalistic type formers $\bar{\tau}$, a term t of $\text{L}_p^{\bar{\tau}}$ is in $\text{Lm}_p^{\bar{\tau}}$, if and only if all of the following hold:

- for any subexpression t_ϵ of t , $|\text{FV}_s(t_\epsilon)| = 0$;
- for any sub-evaluation-context e_ϵ of t , $|\text{FV}_s(e_\epsilon)| = 1$; and
- for any subcommand c of t , $|\text{FV}_s(c)| = 1$ (and this last condition on commands is redundant).

Proof

- \Rightarrow The \leq inequalities are given by Fact V.1.19 and the \geq inequalities by Fact V.2.12.
- \Leftarrow It suffices to remove all right weakening rules in the derivation. More precisely, we show by induction on the derivation that

$$\Gamma \vdash \underline{t_\epsilon : \epsilon} \mid \Delta, \quad (\text{resp. } \Gamma \mid \underline{e_\epsilon : \epsilon} \vdash \Delta, \quad c : (\Gamma \vdash \Delta))$$

implies

$$\Gamma \vdash_m \underline{t_\epsilon : \epsilon} \mid, \quad (\text{resp. } \Gamma \mid \underline{e_\epsilon : \epsilon} \vdash_m \alpha^{\epsilon*} : \epsilon_*, \quad c : (\Gamma \vdash_m \alpha^{\epsilon*} : \epsilon_*))$$

for some $\alpha^{\epsilon*}$. For right weakening rules, we simply apply the induction hypothesis to the premise, and for other rules, we apply the induction hypothesis to the premises and then reapply the same rule. This works for the $(\vdash \tau_-^j)$ (resp. $(\vdash \mu)$) rule because the $|\text{FV}_s(\mu < \dots >)| = 0$ (resp. $|\text{FV}_s(\mu \alpha^{\epsilon*} . c)| = 0$) hypothesis ensures that the free stack variables it binds are exactly those that are free in the subcommands, and for the $(\tau_+^j \vdash)$ rule of types with more than one constructor because the condition $|\text{FV}_s(\tilde{\mu}[\dots])| = 1$ ensures that all the variables $\alpha^{\epsilon*}$ given by the induction hypothesis are the same.

- The condition on commands is redundant because

$$|\text{FV}_s(\langle t_\epsilon | e_\epsilon \rangle^\epsilon)| = |\text{FV}_s(t_\epsilon) \cup \text{FV}_s(e_\epsilon)| = |\emptyset \cup \text{FV}_s(e_\epsilon)| = |\text{FV}_s(e_\epsilon)| = 1$$

Output polarities

The inference rules that define $\text{Lm}_p^{\bar{\tau}}$ can be seen as production rules of a general grammar whose non-terminal symbols are the judgements, and whose terminal symbols are paren-

V. Polarized calculi with arbitrary constructors

theses and rule names: an inference rule

$$\frac{\text{first premise} \quad \dots \quad \text{last premise}}{\text{conclusion}} \text{NAME}$$

becomes a production rule

$$\text{conclusion} \rightarrow \text{name}((\text{first premise}), \dots, (\text{last premise}))$$

This grammar is not really a syntax (i.e. it is not context-free) because there are infinitely many distinct judgements. By Proposition V.2.14, we can discard Γ . This is not sufficient because $\alpha^{\varepsilon_\star}$ ranges over infinitely many names, and we can not discard it because $\mu\beta^\varepsilon.\langle x^{\varepsilon_\star} | \alpha^{\varepsilon_\star} \rangle^{\varepsilon_\star}$ being in $\text{Lm}_p^{\bar{\varepsilon}}$ depends on whether $\alpha^{\varepsilon_\star} = \beta^\varepsilon$ or not. Instead, we switch to a presentation where stack variable names α are replaced by de Bruijn indices \star_0, \star_1 , and so on, and we write \star for \star_0 . With this presentation, judgements are of the shape

$$\Gamma \vdash \underline{t_\varepsilon : \varepsilon} \mid, \quad \Gamma \mid \underline{e_\varepsilon : \varepsilon} \vdash \star^{\varepsilon_\star} : \varepsilon_\star, \quad \text{or} \quad c : (\Gamma \vdash \star^{\varepsilon_\star} : \varepsilon_\star)$$

and $\mu\beta^\varepsilon.\langle x^{\varepsilon_\star} | \star^{\varepsilon_\star} \rangle^{\varepsilon_\star}$ is in $\text{Lm}_p^{\bar{\varepsilon}}$ if and only if $\beta^\varepsilon = \star^{\varepsilon_\star}$. By erasing Γ and replacing it by $\textcircled{?}$, we get a finite set of judgements:

$$\begin{array}{ll} \textcircled{?} \vdash \underline{t_+ : +} \mid & \textcircled{?} \vdash \underline{t_- : -} \mid \\ c : (\textcircled{?} \vdash \star^+ : +) & c : (\textcircled{?} \vdash \star^- : -) \\ \textcircled{?} \mid \underline{e_+ : +} \vdash \star^+ : + & \textcircled{?} \mid \underline{e_+ : +} \vdash \star^- : - \\ \textcircled{?} \mid \underline{e_- : -} \vdash \star^+ : + & \textcircled{?} \mid \underline{e_- : -} \vdash \star^- : - \end{array}$$

We introduce concise notations that allow making explicit which one of these judgements holds for the term under consideration:

Definition V.2.15

Given an evaluation context e_ε (resp. command c) of $\text{Lm}_p^{\bar{\varepsilon}}$, we say that it has *output polarity* ε_\star when there exists a derivation of

$$c : (\Gamma \vdash \star^{\varepsilon_\star} : \varepsilon_\star) \quad (\text{resp. } \Gamma \mid \underline{e_\varepsilon : \varepsilon} \vdash \star^{\varepsilon_\star} : \varepsilon_\star)$$

for some Γ . We write $e_{\varepsilon \rightarrow \varepsilon_\star}$ (resp. $c_{\rightarrow \varepsilon_\star}$) for evaluation contexts e_ε (resp. commands c) of output polarity ε_\star . We call the polarity ε of a term t_ε (resp. evaluation context e_ε) its *interaction polarity*, and sometimes also call the interaction polarity ε of an evaluation context $e_{\varepsilon \rightarrow \varepsilon_\star}$ its *input polarity*.

A BNF grammar for $\text{Lm}_p^{\bar{\varepsilon}}$

Figure V.2.2: The Lm_p^τ calculus

Figure V.2.2.a: Syntax

Negative values / expressions:

$$\begin{aligned}
v_-, w_- &::= x^- \mid \mu \star^- . c_{\rightarrow -} \\
&\mid \mu \left\langle \mathfrak{B}_1^{\tau_1}(\overrightarrow{x_1}, \star^{\varepsilon_{1,1}}) . c_{\rightarrow \varepsilon_{1,1}}^1 \mid \dots \mid \mathfrak{B}_{l_1^-}^{\tau_1}(\overrightarrow{x_{l_1^-}}, \star^{\varepsilon_{1,l_1^-}}) . c_{\rightarrow \varepsilon_{1,l_1^-}}^{l_1^-} \right\rangle \\
&\mid \vdots \\
&\mid \mu \left\langle \mathfrak{B}_1^{\tau_m}(\overrightarrow{x_1}, \star^{\varepsilon_{m,1}}) . c_{\rightarrow \varepsilon_{m,1}}^1 \mid \dots \mid \mathfrak{B}_{l_m^-}^{\tau_m}(\overrightarrow{x_{l_m^-}}, \star^{\varepsilon_{m,l_m^-}}) . c_{\rightarrow \varepsilon_{m,l_m^-}}^{l_m^-} \right\rangle
\end{aligned}$$

Positive values:

$$\begin{aligned}
v_+, w_+ &::= x^+ \\
&\mid \mathfrak{b}_1^{\tau_1}(\overrightarrow{v}) \mid \dots \mid \mathfrak{b}_{l_1^+}^{\tau_1}(\overrightarrow{v}) \\
&\mid \vdots \quad \mid \ddots \mid \vdots \\
&\mid \mathfrak{b}_1^{\tau_n}(\overrightarrow{v}) \mid \dots \mid \mathfrak{b}_{l_n^+}^{\tau_n}(\overrightarrow{v})
\end{aligned}$$

Positive expressions:

$$t_+, u_+ ::= \text{val}^+(v_+) \mid \mu \star^+ . c_{\rightarrow +}$$

Commands:

$$c_{\rightarrow \varepsilon} ::= \langle t_+ \mid e_{\rightarrow + \varepsilon} \rangle^+ \mid \langle t_- \mid e_{\rightarrow - \varepsilon} \rangle^-$$

Negative stacks:

$$\begin{aligned}
s_{\rightarrow \varepsilon} &::= \star^- \}^{\varepsilon=-} \\
&\mid \mathfrak{B}_1^{\tau_1}(\overrightarrow{v}, s_{\varepsilon_{1,1} \rightarrow \varepsilon}) \mid \dots \mid \mathfrak{B}_{l_1^-}^{\tau_1}(\overrightarrow{v}, s_{\varepsilon_{1,l_1^-} \rightarrow \varepsilon}) \\
&\mid \vdots \quad \mid \ddots \mid \vdots \\
&\mid \mathfrak{B}_1^{\tau_m}(\overrightarrow{v}, s_{\varepsilon_{m,1} \rightarrow \varepsilon}) \mid \dots \mid \mathfrak{B}_{l_m^-}^{\tau_m}(\overrightarrow{v}, s_{\varepsilon_{m,l_m^-} \rightarrow \varepsilon})
\end{aligned}$$

Negative evaluation contexts:

$$e_{\rightarrow \varepsilon} ::= \text{stk}^-(s_{\rightarrow \varepsilon}) \mid \tilde{\mu} x^- . c_{\rightarrow \varepsilon}$$

Positive stacks / evaluation contexts:

$$\begin{aligned}
s_{\rightarrow + \varepsilon}, e_{\rightarrow + \varepsilon} &::= \star^+ \}^{\varepsilon=+} \mid \tilde{\mu} x^+ . c_{\rightarrow \varepsilon} \\
&\mid \tilde{\mu} \left[\mathfrak{b}_1^{\tau_1}(\overrightarrow{x_1}) . c_{\rightarrow \varepsilon}^1 \mid \dots \mid \mathfrak{b}_{l_1^+}^{\tau_1}(\overrightarrow{x_{l_1^+}}) . c_{\rightarrow \varepsilon}^{l_1^+} \right] \\
&\mid \vdots \\
&\mid \tilde{\mu} \left[\mathfrak{b}_1^{\tau_n}(\overrightarrow{x_1}) . c_{\rightarrow \varepsilon}^1 \mid \dots \mid \mathfrak{b}_{l_n^+}^{\tau_n}(\overrightarrow{x_{l_n^+}}) . c_{\rightarrow \varepsilon}^{l_n^+} \right]
\end{aligned}$$

Figure V.2.2.b: Operational reduction

$$\begin{aligned}
& \langle \mu \star^\varepsilon . c_{\rightarrow \varepsilon} | s_\varepsilon \rangle^\varepsilon \triangleright_\mu c_{\rightarrow \varepsilon} [s_\varepsilon / \star^\varepsilon] \\
& \langle v_{\varepsilon_1} | \tilde{\mu} x^{\varepsilon_1} . c_{\rightarrow \varepsilon_2} \rangle^{\varepsilon_1} \triangleright_{\tilde{\mu}} c_{\rightarrow \varepsilon_2} [v_{\varepsilon_1} / x^{\varepsilon_1}] \\
& \left\langle \mu \left\langle \mathfrak{s}_1^{\tau_1^j}(\vec{x}_1, \star^{\varepsilon_{j,1}}) . c_{\rightarrow \varepsilon_{j,1}}^1 | \dots | \mathfrak{s}_l^{\tau_l^j}(\vec{x}_l, \star^{\varepsilon_{j,l}}) . c_{\rightarrow \varepsilon_{j,l}}^l \right\rangle \left| \mathfrak{s}_k^{\tau_k^j}(\vec{v}, s_{\varepsilon_{j,k} \rightarrow \varepsilon}) \right\rangle^- \right\rangle \triangleright_{\tau_-^j} c_{\rightarrow \varepsilon_{j,k}}^k [\vec{v} / \vec{x}_k, s_{\varepsilon_{j,k} \rightarrow \varepsilon} / \star^{\varepsilon_{j,k}}] \\
& \left\langle \mathfrak{b}_k^{\tau_k^j}(\vec{v}) \left| \tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^j}(\vec{x}_1) . c_{\rightarrow \varepsilon}^1 | \dots | \mathfrak{b}_l^{\tau_l^j}(\vec{x}_l) . c_{\rightarrow \varepsilon}^l \right]^+ \right\rangle \right\rangle \triangleright_{\tau_+^j} c_{\rightarrow \varepsilon}^k [\vec{v} / \vec{x}_k] \\
& \triangleright \stackrel{\text{def}}{=} \triangleright_{\tilde{\mu}} \cup \triangleright_{\mu} \cup \left(\bigcup_j \triangleright_{\tau_-^j} \right) \cup \left(\bigcup_j \triangleright_{\tau_+^j} \right)
\end{aligned}$$

Figure V.2.2.c: Top-level η -expansion

$$\begin{aligned}
& t_\varepsilon \Downarrow_\mu \mu \star^\varepsilon . \langle t_\varepsilon | \star^\varepsilon \rangle^\varepsilon \\
& e_{\varepsilon_1 \rightarrow \varepsilon_2} \Downarrow_{\tilde{\mu}} \tilde{\mu} x^{\varepsilon_1} . \langle x^{\varepsilon_1} | e_{\varepsilon_1 \rightarrow \varepsilon_2} \rangle^{\varepsilon_1} \quad \text{if } x^{\varepsilon_1} \text{ fresh w.r.t. } e_{\varepsilon_1 \rightarrow \varepsilon_2} \\
& v_- \Downarrow_{\tau_-^j} \mu \left\langle \mathfrak{s}_1^{\tau_1^j}(\vec{x}_1, \star^{\varepsilon_{j,1}}) . \langle v_- | \mathfrak{s}_1^{\tau_1^j}(\vec{x}_1, \star^{\varepsilon_{j,1}}) \rangle^- \right. \\
& \quad \vdots \\
& \quad \left. \mathfrak{s}_l^{\tau_l^j}(\vec{x}_l, \star^{\varepsilon_{j,l}}) . \langle v_- | \mathfrak{s}_l^{\tau_l^j}(\vec{x}_l, \star^{\varepsilon_{j,l}}) \rangle^- \right\rangle \quad \text{if } \vec{x}_1, \dots, \vec{x}_l \text{ fresh w.r.t. } v_- \\
& s_{+ \rightarrow \varepsilon} \Downarrow_{\tau_+^j} \tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^j}(\vec{x}_1) . \langle \mathfrak{b}_1^{\tau_1^j}(\vec{x}_1) | s_{+ \rightarrow \varepsilon} \rangle^+ \right. \\
& \quad \vdots \\
& \quad \left. \mathfrak{b}_l^{\tau_l^j}(\vec{x}_l) . \langle \mathfrak{b}_l^{\tau_l^j}(\vec{x}_l) | s_{+ \rightarrow \varepsilon} \rangle^+ \right] \quad \text{if } \vec{x}_1, \dots, \vec{x}_l \text{ fresh w.r.t. } s_{+ \rightarrow \varepsilon} \\
& \Downarrow \stackrel{\text{def}}{=} \Downarrow_\mu \cup \Downarrow_{\tilde{\mu}} \cup \left(\bigcup_j \Downarrow_{\tau_-^j} \right) \cup \left(\bigcup_j \Downarrow_{\tau_+^j} \right)
\end{aligned}$$

Figure V.2.3: The $\text{Lm}_p^{\rightarrow \Downarrow \uparrow \otimes \oplus \& 10^\top}$ calculus

Figure V.2.3.a: Syntax

Negative values / expressions:

$$\begin{aligned}
v_-, w_- ::= & x^- \mid \mu \star^-. c_{\rightarrow -} \\
& \mid \mu(x^+ \cdot \star^-). c_{\rightarrow -} \\
& \mid \mu \langle (\pi_1 \cdot \star^-). c_{\rightarrow -}^1 \mid (\pi_2 \cdot \star^-). c_{\rightarrow -}^2 \rangle \\
& \mid \mu \{ \alpha^+ \}. c \\
& \mid \mu \langle \rangle
\end{aligned}$$

Positive values:

$$\begin{aligned}
v_+, w_+ ::= & x^+ \\
& \mid (v_+ \otimes w_+) \\
& \mid \iota_1(v_+) \mid \iota_2(v_+) \\
& \mid \{v_-\} \\
& \mid ()
\end{aligned}$$

Positive expressions:

$$t_+, u_+ ::= \text{val}^+(v_+) \mid \mu \star^+. c_{\rightarrow +}$$

Commands:

$$c_{\rightarrow \varepsilon} ::= \langle t_+ \mid e_{\rightarrow \varepsilon} \rangle^+ \mid \langle t_- \mid e_{\rightarrow \varepsilon} \rangle^-$$

Negative stacks:

$$\begin{aligned}
s_{\rightarrow \varepsilon} ::= & \star^- \}^{\varepsilon=-} \\
& \mid v_+ \cdot s_{\rightarrow \varepsilon} \\
& \mid \pi_1 \cdot s_{\rightarrow \varepsilon} \mid \pi_2 \cdot s_{\rightarrow \varepsilon} \\
& \mid \{s_{\rightarrow \varepsilon}\}
\end{aligned}$$

Negative evaluation contexts:

$$e_{\rightarrow \varepsilon} ::= \text{stk}^-(s_{\rightarrow \varepsilon}) \mid \tilde{\mu} x^-. c_{\rightarrow \varepsilon}$$

Positive stacks / evaluation contexts:

$$\begin{aligned}
s_{\rightarrow \varepsilon}, e_{\rightarrow \varepsilon} ::= & \star^+ \}^{\varepsilon=+} \mid \tilde{\mu} x^+. c_{\rightarrow \varepsilon} \\
& \mid \tilde{\mu}(x^+ \otimes y^+). c_{\rightarrow \varepsilon} \\
& \mid \tilde{\mu}[\iota_1(x_1^+). c_{\rightarrow \varepsilon}^1 \mid \iota_2(x_2^+). c_{\rightarrow \varepsilon}^2] \\
& \mid \tilde{\mu}\{x^-\}. c_{\rightarrow \varepsilon} \\
& \mid \tilde{\mu}(). c_{\rightarrow \varepsilon}
\end{aligned}$$

Figure V.2.3.b: Operational reduction

$$\begin{aligned}
& \langle \mu \star^\varepsilon . c_{\rightarrow \varepsilon} | s_\varepsilon \rangle^\varepsilon \triangleright_\mu c_{\rightarrow \varepsilon} [s_\varepsilon / \star^\varepsilon] \\
& \langle v_{\varepsilon_1} | \tilde{\mu} x^{\varepsilon_1} . c_{\rightarrow \varepsilon_2} \rangle^{\varepsilon_1} \triangleright_{\tilde{\mu}} c_{\rightarrow \varepsilon_2} [v_{\varepsilon_1} / x^{\varepsilon_1}] \\
& \langle \mu(x^+ \cdot \star^-) . c_{\rightarrow -} | v_+ \cdot s_{\rightarrow \varepsilon} \rangle^- \triangleright_{\rightarrow} c_{\rightarrow -} [v_+ / x^+, s_{\rightarrow \varepsilon} / \star^-] \\
& \langle \mu \{ \star^+ \} . c_{\rightarrow +} | \{ s_{\rightarrow \varepsilon} \} \rangle^- \triangleright_{\uparrow} c_{\rightarrow +} [s_{\rightarrow \varepsilon} / \star^+] \\
& \langle \mu \langle (\pi_1 \cdot \star^-) . c_{\rightarrow -}^1 | (\pi_2 \cdot \star^-) . c_{\rightarrow -}^2 \rangle | \pi_i \cdot s_{\rightarrow \varepsilon} \rangle^- \triangleright_{\&} c_{\rightarrow -}^i [s_{\rightarrow \varepsilon} / \star^-] \\
& \quad (\triangleright_{\top} \text{ is trivial}) \\
& \langle \{ v_- \} | \tilde{\mu} \{ x^- \} . c_{\rightarrow \varepsilon} \rangle^+ \triangleright_{\downarrow} c_{\rightarrow \varepsilon} [v_- / x^-] \\
& \langle (v_+ \otimes w_+) | \tilde{\mu} (x^+ \otimes y^+) . c_{\rightarrow \varepsilon} \rangle^+ \triangleright_{\otimes} c_{\rightarrow \varepsilon} [v_+ / x^+, w_+ / y^+] \\
& \langle \iota_i(v_+) | \tilde{\mu} [\iota_1(x_1^+) . c_{\rightarrow \varepsilon}^1 | \iota_2(x_2^+) . c_{\rightarrow \varepsilon}^2] \rangle^+ \triangleright_{\oplus} c_{\rightarrow \varepsilon}^i [v_+ / x_i^+] \\
& \langle () | \tilde{\mu} () . c_{\rightarrow \varepsilon} \rangle^+ \triangleright_1 c_{\rightarrow \varepsilon} \\
& \triangleright \stackrel{\text{def}}{=} \triangleright_{\tilde{\mu}} \cup \triangleright_{\tilde{\mu}} \cup \triangleright_{\rightarrow} \cup \triangleright_{\&} \cup \triangleright_{\uparrow} \cup \triangleright_{\otimes} \cup \triangleright_{\oplus} \cup \triangleright_{\downarrow} \cup \triangleright_1
\end{aligned}$$

Figure V.2.3.c: Top-level η -expansion

$$\begin{aligned}
 & t_\varepsilon \Downarrow_\mu \mu \star^\varepsilon . \langle t_\varepsilon | \star^\varepsilon \rangle^\varepsilon \\
 & e_{\varepsilon_1 \leadsto \varepsilon_2} \Downarrow_{\tilde{\mu}} \tilde{\mu} x^{\varepsilon_1} . \langle x^{\varepsilon_1} | e_{\varepsilon_1 \leadsto \varepsilon_2} \rangle^{\varepsilon_1} \quad \text{if } x^{\varepsilon_1} \text{ fresh w.r.t. } e_{\varepsilon_1 \leadsto \varepsilon_2} \\
 & v_- \Downarrow_{\tau_-^j} \mu \left\langle \mathfrak{b}_1^{\tau_-^j}(\vec{x}_1, \star^{\varepsilon_{j,1}}) . \langle v_- | \mathfrak{b}_1^{\tau_-^j}(\vec{x}_1, \star^{\varepsilon_{j,1}}) \rangle^- \right. \\
 & \quad \left. \begin{array}{c} \vdots \\ \mathfrak{b}_l^{\tau_-^j}(\vec{x}_l, \star^{\varepsilon_{j,l}}) . \langle v_- | \mathfrak{b}_l^{\tau_-^j}(\vec{x}_l, \star^{\varepsilon_{j,l}}) \rangle^- \end{array} \right\rangle^- \quad \text{if } \vec{x}_1, \dots, \vec{x}_l \text{ fresh w.r.t. } v_- \\
 & s_{+\leadsto\varepsilon} \Downarrow_{\tau_+^j} \tilde{\mu} \left[\mathfrak{b}_1^{\tau_+^j}(\vec{x}_1) . \langle \mathfrak{b}_1^{\tau_+^j}(\vec{x}_1) | s_{+\leadsto\varepsilon} \rangle^+ \right. \\
 & \quad \left. \begin{array}{c} \vdots \\ \mathfrak{b}_l^{\tau_+^j}(\vec{x}_l) . \langle \mathfrak{b}_l^{\tau_+^j}(\vec{x}_l) | s_{+\leadsto\varepsilon} \rangle^+ \end{array} \right] \quad \text{if } \vec{x}_1, \dots, \vec{x}_l \text{ fresh w.r.t. } s_{+\leadsto\varepsilon} \\
 & \Downarrow \stackrel{\text{def}}{=} \Downarrow_\mu \cup \Downarrow_{\tilde{\mu}} \cup \left(\bigcup_j \Downarrow_{\tau_-^j} \right) \cup \left(\bigcup_j \Downarrow_{\tau_+^j} \right)
 \end{aligned}$$

V. Polarized calculi with arbitrary constructors

Given a set of straightforwardly minimalistic type formers $\vec{\tau}$, the syntax of the $\text{Lm}_p^{\vec{\tau}}$ calculus is given in Figure V.2.2a, where $\varepsilon_{j,k}$ denotes the (input) polarity of the stack argument of $\mathfrak{s}_{k-}^{\tau_j}$. Note that although it is not explicit in the BNF grammar, the restriction to straightforwardly minimalistic types only allows strictly positive l_j^+ . Instanciated with $\vec{\tau} = \rightarrow \Downarrow \Uparrow \otimes \oplus \& 10 \top$, this yields Figure V.2.3a.

Since the main difference between the syntax of $s_{\varepsilon \rightarrow +}$ and $s_{\varepsilon \rightarrow -}$ is only whether it contains \star^ε or not (with $s_{+ \rightarrow +}$ containing \star^+ , $s_{- \rightarrow -}$ containing \star^- , and neither $s_{+ \rightarrow -}$ nor $s_{- \rightarrow +}$ containing any \star^ε), we avoid duplications by having side conditions in the grammar, e.g.

$$s_{+ \rightarrow \varepsilon} ::= \star^+ \} \varepsilon = +$$

means that $s_{+ \rightarrow +}$ can be \star^+ but $s_{+ \rightarrow -}$ can not. For example,

$$\begin{aligned} s_{+ \rightarrow \varepsilon} ::= \star^+ \} \varepsilon = + \mid & \tilde{\mu} x^+ . c_{\rightarrow \varepsilon} \\ & \mid \left[\tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^+}(\overrightarrow{x_1}) . c_{\rightarrow \varepsilon}^1 \mid \dots \mid \mathfrak{b}_{l_1^+}^{\tau_{l_1^+}^+}(\overrightarrow{x_{l_1^+}}) . c_{\rightarrow \varepsilon}^{l_1^+} \right] \right. \\ & \mid \vdots \\ & \left. \mid \tilde{\mu} \left[\mathfrak{b}_1^{\tau_n^+}(\overrightarrow{x_1}) . c_{\rightarrow \varepsilon}^1 \mid \dots \mid \mathfrak{b}_{l_n^+}^{\tau_{l_n^+}^+}(\overrightarrow{x_{l_n^+}}) . c_{\rightarrow \varepsilon}^{l_n^+} \right] \right] \end{aligned}$$

stands for

$$\begin{aligned} s_{+ \rightarrow +} ::= \star^+ \mid & \tilde{\mu} x^+ . c_{\rightarrow +} \\ & \mid \left[\tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^+}(\overrightarrow{x_1}) . c_{\rightarrow +}^1 \mid \dots \mid \mathfrak{b}_{l_1^+}^{\tau_{l_1^+}^+}(\overrightarrow{x_{l_1^+}}) . c_{\rightarrow +}^{l_1^+} \right] \right. \\ & \mid \vdots \\ & \left. \mid \tilde{\mu} \left[\mathfrak{b}_1^{\tau_n^+}(\overrightarrow{x_1}) . c_{\rightarrow +}^1 \mid \dots \mid \mathfrak{b}_{l_n^+}^{\tau_{l_n^+}^+}(\overrightarrow{x_{l_n^+}}) . c_{\rightarrow +}^{l_n^+} \right] \right] \end{aligned}$$

and

$$\begin{aligned} s_{+ \rightarrow -} ::= & \tilde{\mu} x^+ . c_{\rightarrow -} \\ & \mid \left[\tilde{\mu} \left[\mathfrak{b}_1^{\tau_1^+}(\overrightarrow{x_1}) . c_{\rightarrow -}^1 \mid \dots \mid \mathfrak{b}_{l_1^+}^{\tau_{l_1^+}^+}(\overrightarrow{x_{l_1^+}}) . c_{\rightarrow -}^{l_1^+} \right] \right. \\ & \mid \vdots \\ & \left. \mid \tilde{\mu} \left[\mathfrak{b}_1^{\tau_n^+}(\overrightarrow{x_1}) . c_{\rightarrow -}^1 \mid \dots \mid \mathfrak{b}_{l_n^+}^{\tau_{l_n^+}^+}(\overrightarrow{x_{l_n^+}}) . c_{\rightarrow -}^{l_n^+} \right] \right] \end{aligned}$$

Fact V.2.16

The grammar given in Figure V.2.2a describes exactly all minimalistic terms.

Proof

By Proposition V.2.14 and induction on the term.

Remark V.2.17

For $\text{Li}_p^{\vec{\tau}}$, there are three additional kinds of judgements

$$c : (\mathbf{?} \vdash), \quad \mathbf{?} \mid \underline{e_+ : +} \vdash, \quad \text{and} \quad \mathbf{?} \mid \underline{e_- : -} \vdash$$

Applying the same method as for $\text{Lm}_p^{\vec{\tau}}$, we could introduce extra non-terminal symbols $c_{\rightarrow \emptyset}$, $e_{+\rightarrow \emptyset}$, and $e_{-\rightarrow \emptyset}$ for the judgements above. This would yield a grammar that tracks the uses of right weakening rules, which makes it ambiguous: there are two derivations

$$c_{\rightarrow \varepsilon} \rightarrow \tilde{\mu} \left[\mathbf{b}_1^{\tau_1^j}(\vec{x}_1, \star^\varepsilon).c_{\rightarrow \varepsilon}^1 \mid \dots \mid \mathbf{b}_l^{\tau_l^j}(\vec{x}_l, \star^\varepsilon).c_{\rightarrow \varepsilon}^l \right] \rightarrow^* \tilde{\mu} \left[\mathbf{b}_1^{\tau_1^j}(\vec{x}_1, \star^\varepsilon).c_{\rightarrow \emptyset}^1 \mid \dots \mid \mathbf{b}_l^{\tau_l^j}(\vec{x}_l, \star^\varepsilon).c_{\rightarrow \emptyset}^l \right]$$

and

$$c_{\rightarrow \varepsilon} \rightarrow c_{\rightarrow \emptyset} \rightarrow \tilde{\mu} \left[\mathbf{b}_1^{\tau_1^j}(\vec{x}_1, \star^\varepsilon).c_{\rightarrow \emptyset}^1 \mid \dots \mid \mathbf{b}_l^{\tau_l^j}(\vec{x}_l, \star^\varepsilon).c_{\rightarrow \emptyset}^l \right]$$

that correspond to applying $(\vdash \text{wt})$ before and after $(\tau_+^j \vdash)$ respectively. This grammar can be made non-ambiguous by explicitly tracking the free stack variables, but the resulting grammar would be fairly tedious to work with.

V.2.3. Properties

Disubstitutions

In $\text{Lm}_p^{\vec{\tau}}$, we are only interested in some disubstitutions:

Definition V.2.18

A disubstitution is said to be *minimalistic* if its image is contained in $\text{Lm}_p^{\vec{\tau}}$, and it acts non-trivially on at most one stack variable \star^ε .

The $\text{Lm}_p^{\vec{\tau}}$ calculus is closed under minimalistic disubstitutions:

Fact V.2.19

For any minimalistic term t and minimalistic disubstitution φ , $o[\varphi]$ is minimalistic (resp. intuitionistic).

Proof

By induction on t .

Since expressions have no free stacks variables, a minimalistic disubstitution can always be written as the composition of a value substitution and a stack substitution:

Fact V.2.20

For any minimalistic disubstitution $\varphi = \sigma, \star^{\varepsilon_1} \mapsto s_{\varepsilon_1 \leadsto \varepsilon_2}$ and minimalistic term t , we have

$$t[\varphi] = t[\sigma][s_{\varepsilon_1 \leadsto \varepsilon_2} / \star^{\varepsilon_1}]$$

Proof

By induction on t . The base case $t = \star^{\varepsilon}$ is immediate, the base case $t = x^{\varepsilon}$ boils down to the fact that $x^{\varepsilon}[\sigma]$ is an expression and therefore has no stack variable, and in all the remaining cases, the induction hypothesis immediately allows to conclude.

Reductions

Descriptions of the restriction of \triangleright and \downarrow to $\text{Lm}_p^{\bar{\tau}}$ are given in Figures V.2.2b and V.2.2c. Note in particular that these only involve minimalistic disubstitutions, and $\text{Lm}_p^{\bar{\tau}}$ is therefore closed under the operational reduction \triangleright , top-level η -expansion \downarrow , and η -reduction \downarrow and their respective contextual closures \rightarrow , \rightarrow , and \vdash :

Fact V.2.21: Closure of $\text{Lm}_p^{\bar{\tau}}$ under \vdash

If $t \vdash t'$ and t is minimalistic then so is t' .

Proof

Closure under \triangleright follows from closure under minimalistic disubstitution (Fact V.2.19). Closure under \downarrow and \downarrow is immediate. Closure under their contextual closures \rightarrow , \rightarrow and \vdash follows by induction on the derivation.

Thanks to this closure property, disubstitutivity, confluence, postponement and factorization transfer from $\text{L}_p^{\bar{\tau}}$ to $\text{Lm}_p^{\bar{\tau}}$:

Fact V.2.22

In $\text{Lm}_p^{\bar{\tau}}$, the reductions $\triangleright, \downarrow, \rightarrow, \rightarrow$ are disubstitutive.

Proof

Suppose that t is a minimalistic term such that that $t \rightsquigarrow t'$ for some reduction $\rightsquigarrow \in \{\triangleright, \downarrow, \rightarrow, \rightarrow\}$, and let φ be a minimalistic (resp. intuitionistic) disubstitution. By disubstitutivity in $\text{L}_p^{\bar{\tau}}$ of \rightsquigarrow , we have $c[\varphi] \rightsquigarrow c'[\varphi]$. By Fact V.2.19, $c[\varphi]$ is minimalistic, and by Fact V.2.21 so is $c'[\varphi]$.

V. Polarized calculi with arbitrary constructors

Proposition V.2.23: Confluence of \multimap in $\text{Lm}_p^{\bar{\tau}}$

In $\text{Lm}_p^{\bar{\tau}}$, \multimap is confluent .

Proof

By confluence in $L_p^{\bar{\tau}}$ (Proposition ??) and closure of $\text{Lm}_p^{\bar{\tau}}$ under \multimap (Fact V.2.21).

Proposition V.2.24: Postponement of \multimap^0 after \triangleright in $\text{Lm}_p^{\bar{\tau}}$

In $\text{Lm}_p^{\bar{\tau}}$, \multimap^0 postpones after \triangleright : if $t \multimap^* t'$ then $t \triangleright^* \multimap^0 \triangleright^* t'$.

Proof

By postponement in $L_p^{\bar{\tau}}$ (Proposition ??) and closure of $\text{Lm}_p^{\bar{\tau}}$ under \multimap (Fact V.2.21).

Proposition V.2.25: Factorization of \multimap^* as $\triangleright^* \multimap^0 \triangleright^*$ in $\text{Lm}_p^{\bar{\tau}}$

In $\text{Lm}_p^{\bar{\tau}}$, \multimap^* factorizes as $\multimap^* = \triangleright^* \multimap^0 \triangleright^*$.

Proof

By factorization in $L_p^{\bar{\tau}}$ (Proposition ??) and closure of $\text{Lm}_p^{\bar{\tau}}$ under \multimap (Fact V.2.21).

V.3. A polarized λ -calculus with focus equivalent to $\mathbf{Lm}_{\mathbf{p}}^{\vec{\tau}}$: $\lambda_{\mathbf{p}}^{\vec{\tau}}$

V.4. Equivalence between $\lambda_{\text{p}}^{\vec{\tau}}$ and $\text{Lm}_{\text{p}}^{\vec{\tau}}$

V.5. A polarized λ -calculus: $\lambda_{\text{p}}^{\vec{\tau}}$

VI. Dynamically typed polarized calculi

VI.1. Clashes and dynamically typed calculi

VI.2. A dynamically typed polarized λ -calculus: $\lambda_{\mathbf{p}}^{\mathcal{PN}}$

VI.3. A dynamically typed polarized λ -calculus with focus:

$\lambda_{\text{p}}^{\mathcal{PN}}$


VI.4. A dynamically typed polarized intuitionistic L calculus:

$\mathbf{Li}_p^{\mathcal{PN}}$

VI.5. A dynamically typed polarized classical L calculus: $L_p^{\mathcal{PN}}$

Part C.

Solvability in polarized calculi

Part C is about two well-known and very useful properties of λ -terms: operational relevance and solvability. 

Most common definitions of solvability are optimized to make proofs easier, which has the unfortunate consequence of making it look like a fairly arbitrary notion that just happens to have some use cases. This is of course not the case, and in this introduction we aim at explaining why solvability is a very natural and useful notion when looking at semantics of programming languages. In the λ -calculus, it is well known that solvable expressions are exactly the operationally relevant one and, as will be explained in the next section, this can be understood as saying (somewhat indirectly) that the output of programs can be used internally, i.e. as an intermediate result in a larger program.

Content 

Contribution 

Introduction to solvability and operational completeness

Goal and content The goal of this section is to explain what solvability and operational relevance are, and why they are relevant to the study of real-world programming languages. The common definitions of these notions in the λ -calculus are very specialized, and some only describe meaningful notions because of non-trivial properties of the λ -calculus. We therefore give slightly more general definitions that are easier to motivate, and equivalent to the standard definitions when instantiated in the λ -calculus (see Section ??). We then use various fragments of the OCaml programming language to illustrate these definitions, and demonstrate how some desirable properties can be broken and restored by slightly modifying the programming language.

Caveats Definitions in the general setting may be slightly naive. Indeed, while formally studying the extensions of the common definitions of the λ -calculus to a large class of programming languages is a very interesting perspective (because having several models makes identifying the “nice” definitions much easier¹), this is not our goal here: the goal of this section is only to provide some intuition. Furthermore, in this section, only definitions should be considered formal. In particular, even though vague proof sketches are sometimes provided, any claim made between here and the start of Chapter VII should be understood as being an informal and vague idea, and not a precise formal claim². Indeed, ensuring that the claims about the fragments of OCaml hold might require further restricting the definition of the main fragment OCaml_ℤ in Example C.2³, and formally proving them would require a huge amount of work; but neither of these would be beneficial to our goal of providing intuition, so we keep things informal and approximative.

Summary

C.1. A meaning for programs

Programs as maps from inputs to outputs The definitions of programming languages take various fairly different shapes (ranging from “whatever the compiler / interpreter can handle” to full formal specifications), all of which describe both a set of programs \mathcal{P}_{og} and a

¹For example, if we only consider the ring of integers \mathbb{Z} instead of all (commutative ordered) rings (just like we tend to consider only the λ -calculus instead of many calculi), we have an equivalence between being a neutral element for addition and being strictly between -1 and 1 (just like we have an equivalence between being operationally relevant and being solvable):

$$\forall m, n \quad n + m = m \iff -1 < n < 1$$

Since both properties are equivalent in \mathbb{Z} , one might reasonably prefer using the simplest one, i.e. the right hand one. Instantiating both properties in other rings makes it clear that the two properties are not equivalent in general (e.g. the equivalence fails in the ring of rational numbers \mathbb{Q}), and that the left hand property is “nicer”.

²This is emphasized by all of these claims being made either in the main text, in an example, or in a remark; but never in a fact, proposition, or theorem environment.

³For example, the complexity of type inference might break some definitions or properties, so one might have to require explicit type annotations everywhere and forbid GADTs and phantom types in OCaml_ℤ.

way to interpret a program $p \in \mathcal{P}_{\text{og}}$ as a partial function from inputs to outputs

$$\llbracket p \rrbracket : \mathcal{I}_{\text{input}} \rightarrow \mathcal{O}_{\text{output}}$$

This partial function can often be defined as

$$\llbracket p \rrbracket : \mathcal{I}_{\text{input}} \rightarrow \mathcal{O}_{\text{output}} \\ i \mapsto \begin{cases} \text{output}(q) & \text{if } \text{initial_state}(p, i) \triangleright^* q \triangleright \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\mathcal{S}_{\text{state}}$ is a set of states equipped with

- a deterministic reduction

$$\triangleright \subseteq \mathcal{S}_{\text{state}} \times \mathcal{S}_{\text{state}}$$

that represents evaluation, sometimes called the operational reduction;

- a function

$$\text{initial_state} : \mathcal{P}_{\text{og}} \times \mathcal{I}_{\text{input}} \rightarrow \mathcal{S}_{\text{state}}$$

that combines a program and the input we want to run it into something that can be evaluated, i.e. a state; and

- a partial function

$$\text{output} : \{q \in \mathcal{S}_{\text{state}} : q \triangleright\} \rightarrow \mathbf{Output}$$

recovers the output from the state at the end of the evaluation.

Example C.1

For real-world programming languages, one can think of $\mathcal{S}_{\text{state}}$ as being the set of all possible states of the computer, of $\text{initial_state}(p, i)$ as compiling the program p and placing the result of the compilation alongside the input i at the correct memory locations to make the computer execute the program p with input i , of \triangleright as representing the progress made during one tick of the processor, and of the output function as reading the part of memory that contains the output of the program.

Example C.2

We define $\text{OCaml}_{\mathbb{Z}}$ as the programming language whose programs are OCaml expressions^a of type $\mathbb{Z}.t \rightarrow \mathbb{Z}.t$ with no free variables, where $\mathbb{Z}.t$ is the type of unbounded integers provided by the `Zarith` library^b. An $\text{OCaml}_{\mathbb{Z}}$ program p is evaluated on an input $i \in \mathbb{Z}$ by wrapping it in the boilerplate code described in Figure C.1^c, storing the result in a file `program.ml`, and then running the Bash command

```
ocamlbuild -use-ocamlfind -package zarith program.byte -- i
```

We leave implicit some coercions `Z.of_int` (even though the OCaml compiler does not like that), e.g. writing

```
fun n → n + 1    for    fun n → n + Z.of_int 1
```

To make reasoning about the programs easier, we slightly restrict how the programs p of $\text{OCaml}_{\mathbb{Z}}$ can be written:

- Variable shadowing is disallowed for every name used in the `Pervasives` and `Z` modules. This for example ensures that the symbols $+$, $-$, $*$ and $/$ always refer to the corresponding arithmetic operations on `Z.t`.
- Using non-OCaml code through bindings is forbidden, except `Sys.argv` and those provided by the `Z` module. This ensures that the behavior of the program is not affected by its environment (i.e. the OS, the file system, etc.), and that abstractions can not be broken^d.

Each program p represents a partial function $\llbracket p \rrbracket : \mathbb{Z} \rightarrow \mathbb{Z}$. For example,

$$\llbracket \text{fun } n \rightarrow n \rrbracket = \llbracket \text{fun } n \rightarrow n + 2 - 2 \rrbracket = \llbracket \text{fun } n \rightarrow n * 2 / 2 \rrbracket = \text{Id}_{\mathbb{Z}}$$

but

$$\llbracket \text{fun } n \rightarrow n * n / n \rrbracket = \text{Id}_{\mathbb{Z} \setminus \{0\}}$$

i.e.

$$\llbracket \text{fun } n \rightarrow n * n / n \rrbracket(i) = \begin{cases} i & \text{if } i \neq 0 \\ \text{undefined} & \text{if } i = 0 \end{cases}$$

because the program `fun n → n * n / n` raises a `Division_by_zero` exception on input 0. Recall that in OCaml, those exceptions can be caught by using a try-with statement, e.g.

$$\llbracket \text{fun } n \rightarrow \text{try } n * n / n \text{ with } \text{Division_by_zero} \rightarrow 9 \rrbracket(0) = 9$$



We write $\text{OCaml}_{\mathbb{Z}}^{\neg\text{try}}$ for the fragment of $\text{OCaml}_{\mathbb{Z}}$ with no try-with statements.

^aThe syntax of OCaml expressions is given at <https://ocaml.org/manual/expr.html>

^bWe use the type of unbounded integers `Z.t` from the `Zarith` library instead of the type `int` to avoid any subtleties due to overflows.

^cThe boilerplate checks that the program was given a single argument `input_str`, converts it to an unbounded integer `input`, computes the output by running the `main` function on the `input`, and finally prints the output.

^dFor example, using `Obj.magic`, one can observe values of unknown type in polymorphic functions, hence breaking parametricity.

There are many ways to instantiate the notions in λ -calculi, but for now we only use a call-by-name one and a call-by-value one (see  and  for more instantiations, and a discussion of their relationship):

Example C.3

In the call-by-name λ -calculus $\lambda_{\mathbb{N}}^{\rightarrow}$, we define programs as being closed terms, inputs as being closed stacks (i.e. stacks $\mathbb{S}_N = \square U_N^1 \dots U_N^q$ with each U_N^k closed), the operational reduction \blacktriangleright is the weak head reduction \triangleright_N , and the set of outputs is trivial (i.e.

Figure C.1: An OCaml_Z program p with its implicit boilerplate

```

let main : Z.t → Z.t =
  let open Z in
     $p$ 
  in
  let input =
    match Sys.argv with
    | [| _; input_str |] → Z.of_string_base 10 input_str
    | _ → failwith "Unexpected usage!"
  in
  let output = main input in
  Z.print output

```

it is a singleton $\mathcal{O}_{\text{Output}} = \{\bullet\}$, so that:

$$\begin{aligned}
 \llbracket T_N \rrbracket_N : \overline{\mathbf{S}_N} &\rightarrow \{\bullet\} \\
 \square \vec{U}_N &\mapsto \begin{cases} \bullet & \text{if } T_N \vec{U}_N \triangleright_N^{\otimes} \\ \text{undefined} & \text{if } T_N \vec{U}_N \triangleright_N^{\varepsilon} \end{cases}
 \end{aligned}$$

This corresponds to the so-called lazy λ -calculus [Abr90].

In the call-by-value λ -calculus λ_V^{\rightarrow} , we again define programs as being closed terms, inputs as being closed simple stacks (i.e. simple stacks $\vec{S}_V = \square V_V^1 \dots V_V^q$ with each V_V^k closed), and outputs as being trivial, but the reduction \blacktriangleright is now the call-by-value operational reduction \triangleright_V :

$$\begin{aligned}
 \llbracket T_V \rrbracket_V : \overline{\vec{S}_V} &\rightarrow \{\bullet\} \\
 \square \vec{V}_V &\mapsto \begin{cases} \bullet & \text{if } T_V \vec{V}_V \triangleright_V^{\otimes} \\ \text{undefined} & \text{if } T_V \vec{V}_V \triangleright_V^{\varepsilon} \end{cases}
 \end{aligned}$$

Comparing programs The semantics $\llbracket \cdot \rrbracket$ induces a preorder \lesssim defined by $p_1 \lesssim p_2$ meaning that $\llbracket p_1 \rrbracket$ is a restriction of $\llbracket p_2 \rrbracket$, i.e. that whenever $\llbracket p_1 \rrbracket(I)$ is defined, so is $\llbracket p_2 \rrbracket(I)$ and they are equal:

Definition C.4

The preorder \lesssim is defined on programs by

$$p_1 \lesssim p_2 \stackrel{\text{def}}{=} \llbracket p_1 \rrbracket \subseteq \llbracket p_2 \rrbracket$$

Having $p_1 \lesssim p_2$ means that we can replace p_1 by p_2 without breaking anything: if p_1

computes what we want, then so does p_2 . The preorder in turn induces an equivalence relation \sim and a strict preorder $<$:

- $p_1 \sim p_2$ means that p_1 and p_2 are interchangeable;
- $p_1 < p_2$ means that p_1 can be safely replaced by p_2 while the reverse replacement is not safe.

More formally:

Definition C.5

The equivalence relation \sim and the strict preorder $<$ are defined by:

$$p_1 \sim p_2 \stackrel{\text{def}}{=} \llbracket p_1 \rrbracket = \llbracket p_2 \rrbracket \quad \text{and} \quad p_1 < p_2 \stackrel{\text{def}}{=} \llbracket p_1 \rrbracket \subsetneq \llbracket p_2 \rrbracket$$

Example C.6

In OCaml_ℤ, we have

$$\text{fun } n \rightarrow n * n / n < \text{fun } n \rightarrow n \sim \text{fun } n \rightarrow n + n - n$$

Indeed, since there are no overflows, for inputs $i \neq 0$, we have

$$\llbracket \text{fun } n \rightarrow n * n / n \rrbracket(i) = \llbracket \text{fun } n \rightarrow n \rrbracket(i) = \llbracket \text{fun } n \rightarrow n + n - n \rrbracket(i) = i$$

On input 0, we also have

$$\llbracket \text{fun } n \rightarrow n \rrbracket(0) = \llbracket \text{fun } n \rightarrow n + n - n \rrbracket(0) = 0$$

but

$$\llbracket \text{fun } n \rightarrow n * n / n \rrbracket(0) = \text{undefined}$$

because a `Division_by_zero` exception is raised.

C.2. A compositional meaning for fragments

Fragments and plugging In order to study $\llbracket p \rrbracket$ in a compositional way, we want to look at how it is affected when some fragment of p changes. We therefore assume that the syntax of the programming language is given by a (non-ambiguous) formal grammar (e.g. a BNF / context-free grammar), so that a program p can be represented by a syntax tree, and fragments of p can be represented by (not necessarily downward closed⁴) subtrees of that tree.

More precisely:

⁴ 

Definition C.7

Given a non-ambiguous formal grammar \mathcal{G} , and a non-terminal symbol \mathfrak{A} of \mathcal{G} , we write $\mathcal{T}_{\text{erm}}(\mathfrak{A})$ for the set of syntax tree generated by \mathfrak{A} . We call *term*, and denote by t , any element of the set

$$\mathcal{T}_{\text{erm}} \stackrel{\text{def}}{=} \bigcup_{\mathfrak{A}} \mathcal{T}_{\text{erm}}(\mathfrak{A})$$

Given n non-terminal symbols $\mathfrak{A}_1, \dots, \mathfrak{A}_n$ of \mathcal{G} , we write $\mathcal{G}(\mathfrak{A}_1, \dots, \mathfrak{A}_n)$ for the grammar \mathcal{G} extended by the terminal symbols $\square_{\mathfrak{A}_1}^1, \dots, \square_{\mathfrak{A}_n}^n$, which we call *holes*, and the production rules $\mathfrak{A}_k \rightarrow \square_{\mathfrak{A}_k}^k$. Given $n + 1$ non-terminal symbols $\mathfrak{A}_1, \dots, \mathfrak{A}_n$, and \mathfrak{B} of \mathcal{G} , we write $\mathcal{F}_{\text{rag}}(\mathfrak{A}_1 \otimes \dots \otimes \mathfrak{A}_n, \mathfrak{B})$ for the set of syntax trees generated by \mathfrak{B} in $\mathcal{G}(\mathfrak{A}_1, \dots, \mathfrak{A}_n)$ in which each hole $\square_{\mathfrak{A}_1}^1, \dots, \square_{\mathfrak{A}_n}^n$ occurs exactly once^a. We call *fragment*, and denote by f , any element of the set

$$\mathcal{F}_{\text{rag}} \stackrel{\text{def}}{=} \bigcup_n \bigcup_{\mathfrak{A}_1, \dots, \mathfrak{A}_n} \bigcup_{\mathfrak{B}} \mathcal{F}_{\text{rag}}(\mathfrak{A}_1 \otimes \dots \otimes \mathfrak{A}_n, \mathfrak{B})$$


We call the number n of holes in a fragment its *arity*.

Given two non-terminal symbols \mathfrak{A} and \mathfrak{B} , we define

$$\mathcal{C}_{\text{ontext}}(\mathfrak{A}, \mathfrak{B}) \stackrel{\text{def}}{=} \mathcal{F}_{\text{rag}}(\mathfrak{A}, \mathfrak{B})$$

We call *context*, and denote by \mathbb{C} , any element of the set

$$\mathcal{C}_{\text{ontext}} \stackrel{\text{def}}{=} \bigcup_{\mathfrak{A}} \bigcup_{\mathfrak{B}} \mathcal{C}_{\text{ontext}}(\mathfrak{A}, \mathfrak{B})$$

^aIt would also be reasonable to require the holes $\square_{\mathfrak{A}_1}^1, \dots, \square_{\mathfrak{A}_n}^n$ to appear in this exact order (from left to right). While this would make fragments slightly simpler, it would a priori make some definitions more tedious. For example, “ $\exists f, f[t, \square] \approx \square$ ” (i.e. uniform solvability of a term t , see ) might need to be replaced by “ $\exists f, f[t, \square] \approx \square$ or $\exists f, f[\square, t] \approx \square$ ”.

Since all programs are part of the syntax, we have $\mathcal{P}_{\text{rog}} \subseteq \mathcal{T}_{\text{erm}}$, and this inclusion is often strict:

Example C.8

In OCaml_ℤ, terms are those described by the OCaml syntax, and we have

$$p = \text{fun } m \rightarrow m + 2 \in \mathcal{P}_{\text{rog}} \subseteq \mathcal{T}_{\text{erm}}(\text{expr}) \subseteq \mathcal{T}_{\text{erm}}$$

(because programs are closed OCaml expressions of type $\mathbb{Z}.t \rightarrow \mathbb{Z}.t$), but

$$t_1 = \text{fun } m \rightarrow n + 3 \in \mathcal{T}_{\text{erm}} \setminus \mathcal{P}_{\text{rog}}$$

and

$$t_2 = 4 \in \mathcal{T}_{\text{erm}} \setminus \mathcal{P}_{\text{rog}}$$

because t_1 is not closed and t_2 is not of type $\mathbb{Z}.t \rightarrow \mathbb{Z}.t$. We also have

$$t_3 = \mathbb{Z}.t \in \mathcal{T}_{\text{erm}}(\text{typexpr}) \subseteq \mathcal{T}_{\text{erm}} \setminus \mathcal{P}_{\text{rog}}$$

Definition C.9

We call *plugging* the operation

$$\mathcal{F}_{\text{rag}}\left(\bigotimes_{k=1}^n \mathcal{B}_k, \mathcal{C}\right) \times \mathcal{F}_{\text{rag}}\left(\bigotimes_{j=1}^{m_1} \mathcal{A}_{1,j}, \mathcal{B}_1\right) \times \cdots \times \mathcal{F}_{\text{rag}}\left(\bigotimes_{j=1}^{m_n} \mathcal{A}_{n,j}, \mathcal{B}_n\right) \rightarrow \mathcal{F}_{\text{rag}}\left(\bigotimes_{k=1}^n \bigotimes_{j=1}^{m_k} \mathcal{A}_{k,j}, \mathcal{C}\right)$$

$$(\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_n) \mapsto \mathcal{P}_0 \boxed{\mathcal{P}_1, \dots, \mathcal{P}_n}$$

where $\mathcal{P}_0 \boxed{\mathcal{P}_1, \dots, \mathcal{P}_n}$ denotes the result of simultaneously replacing each hole $\square_{\mathcal{B}_k}^k$ of \mathcal{P}_0 by \mathcal{P}_k (with its holes shifted^a). When $(\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_n)$ is in the domain of this function, we say that $\mathcal{P}_1, \dots, \mathcal{P}_n$ are *pluggable* in \mathcal{P}_0 .

^aGiven fragments $\mathcal{P}_1, \dots, \mathcal{P}_n$ of respective arities m_1, \dots, m_n , we want $\mathcal{P}_0 \boxed{\mathcal{P}_1, \dots, \mathcal{P}_n}$ to have arity $m_1 + \dots + m_n$, with its first m_1 holes corresponding to the holes of \mathcal{P}_1 , ..., and its last m_n holes corresponding to the holes of \mathcal{P}_n . We therefore replace each hole $\square_{\mathcal{A}_{k,j}}^j$ by $\square_{\mathcal{A}_{k,j}}^{m_1 + \dots + m_{k-1} + j}$.

Example C.10

In $\text{OCaml}_{\mathbb{Z}}$,

- $p = \text{fun } m \rightarrow m + 2 \in \mathcal{P}_{\text{og}} \subseteq \mathcal{T}_{\text{erm}}(\text{expr})$,
- $t = n + 3 \in \mathcal{T}_{\text{erm}}(\text{expr}) \setminus \mathcal{P}_{\text{og}}$, and
- $\mathcal{P} = \text{fun } n \rightarrow \square_1 (n * \square_2) \in \mathcal{F}_{\text{rag}}(\text{expr} \otimes \text{expr}, \text{expr})$

can be combined to form the program

$$\mathcal{P} \boxed{p, t} = \text{fun } n \rightarrow (\text{fun } m \rightarrow m + 2) (n * (n + 3)) \in \mathcal{P}_{\text{og}} \subseteq \mathcal{T}_{\text{erm}}(\text{expr})$$

However, t is not pluggable in

$$\mathcal{R} = \text{fun } (m : \square) \rightarrow m + 2 \in \mathcal{C}_{\text{ontext}}(\text{typexpr}, \text{expr})$$

because \mathcal{R} expects a type expression (i.e. an element of $\mathcal{T}_{\text{erm}}(\text{typexpr})$) while t is an expression (i.e. an element of $\mathcal{T}_{\text{erm}}(\text{expr})$).

Plugging is associative, so that fragments (resp. contexts) form a multicategory⁵ (resp. category) with plugging as the composition operation. In particular, given a non-terminal symbol \mathcal{A} , plugging induces a monoid structure on $\mathcal{C}_{\text{ontext}}(\mathcal{A}, \mathcal{A})$ and an action of that monoid on $\mathcal{T}_{\text{erm}}(\mathcal{A})$.

⁵Objects are the non-terminal symbols of the grammar, the sets of morphism are sets of fragments $\mathcal{F}_{\text{rag}}(\mathcal{A}_1 \otimes \dots \otimes \mathcal{A}_n, \mathcal{B})$, and composition is the plugging operation.

Notation C.11

We sometimes abbreviate the names as follows:

$$\mathcal{P} \stackrel{\text{ntn}}{=} \mathcal{P}_{\text{rog}}, \quad \mathcal{T} \stackrel{\text{ntn}}{=} \mathcal{T}_{\text{erm}}, \quad \mathcal{K} \stackrel{\text{ntn}}{=} \mathcal{C}_{\text{ontext}}, \quad \text{and} \quad \mathcal{F} \stackrel{\text{ntn}}{=} \mathcal{F}_{\text{rag}}$$

Program-preserving observational preorder and equivalence We now want to extend \lesssim to a relation \sqsubseteq on fragments that allows us to study \lesssim in a compositional way. Intuitively, $\beta_1 \sqsubseteq \beta_2$ should mean that we can always replace β_1 by β_2 without breaking the surrounding program, i.e that replacing β_1 by β_2 in a program $p_1 = \mathbb{R}[\beta_1 \vec{t}]$ yields a program $p_2 = \mathbb{R}[\beta_2 \vec{t}]$ such that $p_1 \lesssim p_2$.

For both $\mathbb{R}[\beta_2 \vec{t}]$ and $\mathbb{R}[\beta_1 \vec{t}]$ to both make sense, the two fragments must of course be compatible:

Definition C.12

We say that two fragments β_1 and β_2 are *compatible*, and write $\beta_1 \models \beta_2$, when they are in the same set $\mathcal{F}_{\text{rag}}(\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n, \mathcal{B})$:

$$\beta_1 \models \beta_2 \stackrel{\text{def}}{=} \exists n, \exists \mathcal{A}_1, \dots, \exists \mathcal{A}_n, \exists \mathcal{B}, \beta_1 \in \mathcal{F}_{\text{rag}}(\mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n, \mathcal{B}) \ni \beta_2$$

We can then define a relation \sqsubseteq that formalizes the intuition given above as follows:

Definition C.13

The *program-preserving observational preorder* \sqsubseteq is defined on pairs of compatible fragments $(\beta_1, \beta_2) \in \mathcal{F}_{\text{rag}}(\vec{\mathcal{A}}, \mathcal{B})^2$ by

$$\beta_1 \sqsubseteq \beta_2 \stackrel{\text{def}}{=} \forall \mathcal{C}, \forall \mathbb{R} \in \mathcal{K}(\mathcal{B}, \mathcal{C}), \forall \vec{t} \in \mathcal{T}(\vec{\mathcal{A}}), \mathbb{R}[\beta_1 \vec{t}] \in \mathcal{P} \Rightarrow \begin{cases} \mathbb{R}[\beta_2 \vec{t}] \in \mathcal{P}, \text{ and} \\ \mathbb{R}[\beta_1 \vec{t}] \lesssim \mathbb{R}[\beta_2 \vec{t}] \end{cases}$$

(where $\mathcal{T}_{\text{erm}}(\mathcal{A}_1, \dots, \mathcal{A}_n) \stackrel{\text{ntn}}{=} \mathcal{T}_{\text{erm}}(\mathcal{A}_1) \times \cdots \times \mathcal{T}_{\text{erm}}(\mathcal{A}_n)$), and in particular, on terms $(t_1, t_2) \in \mathcal{T}_{\text{erm}}(\mathcal{B})^2$,

$$t_1 \sqsubseteq t_2 \stackrel{\text{def}}{=} \forall \mathcal{C}, \forall \mathbb{R} \in \mathcal{C}_{\text{ontext}}(\mathcal{B}, \mathcal{C}), \mathbb{R}[t_1] \in \mathcal{P}_{\text{rog}} \Rightarrow \begin{cases} \mathbb{R}[t_2] \in \mathcal{P}_{\text{rog}}, \text{ and} \\ \mathbb{R}[t_1] \lesssim \mathbb{R}[t_2] \end{cases}$$

The *program-preserving observational equivalence* \approx is the equivalence relation induced by \sqsubseteq :

$$\beta_1 \approx \beta_2 \stackrel{\text{def}}{=} \beta_1 \sqsubseteq \beta_2 \text{ and } \beta_1 \sqsupseteq \beta_2$$

The *strict program-preserving observational preorder* \sqsubset is the strict preorder induced by \sqsubseteq :

$$\beta_1 \sqsubset \beta_2 \stackrel{\text{def}}{=} \beta_1 \sqsubseteq \beta_2 \text{ but } \beta_1 \not\sqsupseteq \beta_2$$

Remark C.14

It is immediate that

$$\beta_1 \sqsubseteq \beta_2 \Leftrightarrow \forall \vec{t} \in \mathcal{T}_{\text{term}}(\vec{\mathcal{A}}), \beta_1[\vec{t}] \sqsubseteq \beta_2[\vec{t}]$$

so that one may wonder why \sqsubseteq is defined on all fragments and not just on terms. The answer is that it makes the definition more resilient to changes of syntax. For example, if we restricted \sqsubseteq to terms, it would be defined on stacks $s_n = t_n^1 \cdot \dots \cdot t_n^q \cdot \star^n$ of $\text{Li}_{\vec{n}}$ (which are terms t), but not on stacks $s_n = \square T_N^1 \dots T_N^q$ of $\lambda_{\vec{n}}^{\rightarrow}$ (which are contexts \mathcal{R}), even though $\text{Li}_{\vec{n}}$ and $\lambda_{\vec{n}}^{\rightarrow}$ are basically the same calculus.

The program-preserving observational preorder \sqsubseteq (resp. equivalence \approx) can equivalently be expressed as the intersection $\sqsubseteq = \sqsubset \cap \sqsupset$ (resp. $\approx = \sqsubset \cap \sqsupset$) of two simpler relations:

Definition C.15

The *program-preserving preorder* \sqsubset is defined on pairs of compatible fragments $(\beta_1, \beta_2) \in \mathcal{F}_{\text{ag}}(\vec{\mathcal{A}}, \mathcal{B})^2$ by

$$\beta_1 \sqsubset \beta_2 \stackrel{\text{def}}{=} \forall \mathcal{C}, \forall \mathcal{R} \in \mathcal{C}_{\text{context}}(\mathcal{B}, \mathcal{C}), \forall \vec{t} \in \mathcal{T}_{\text{term}}(\vec{\mathcal{A}}), \mathcal{R}[\beta_1[\vec{t}]] \in \mathcal{R}_{\text{og}} \Rightarrow \mathcal{R}[\beta_2[\vec{t}]] \in \mathcal{R}_{\text{og}}$$

The *program-preserving equivalence* \sqsupset is the equivalence relation induced by \sqsubset :

$$\begin{aligned} \beta_1 \sqsupset \beta_2 &\stackrel{\text{def}}{=} \beta_1 \sqsubset \beta_2 \text{ and } \beta_2 \sqsubset \beta_1 \\ &\Leftrightarrow \forall \mathcal{C}, \forall \mathcal{R} \in \mathcal{C}_{\text{context}}(\mathcal{B}, \mathcal{C}), \forall \vec{t} \in \mathcal{T}_{\text{term}}(\vec{\mathcal{A}}), \mathcal{R}[\beta_1[\vec{t}]] \in \mathcal{R}_{\text{og}} \Leftrightarrow \mathcal{R}[\beta_2[\vec{t}]] \in \mathcal{R}_{\text{og}} \end{aligned}$$

Definition C.16

The \lesssim -testing observational relation \sqsubseteq^{\dagger} is defined on pairs of compatible fragments $(\beta_1, \beta_2) \in \mathcal{F}_{\text{ag}}(\vec{\mathcal{A}}, \mathcal{B})^2$ by

$$\beta_1 \sqsubseteq^{\dagger} \beta_2 \stackrel{\text{def}}{=} \forall \mathcal{C}, \forall \mathcal{R} \in \mathcal{C}_{\text{context}}(\mathcal{B}, \mathcal{C}), \forall \vec{t} \in \mathcal{T}_{\text{term}}(\vec{\mathcal{A}}), \left. \begin{array}{l} \mathcal{R}[\beta_1[\vec{t}]] \in \mathcal{R}_{\text{og}} \\ \mathcal{R}[\beta_2[\vec{t}]] \in \mathcal{R}_{\text{og}} \end{array} \right\} \Rightarrow \mathcal{R}[\beta_1[\vec{t}]] \lesssim \mathcal{R}[\beta_2[\vec{t}]]$$

The \sim -testing observational relation \approx^{\dagger} is the symmetric interior of \sqsubseteq^{\dagger} :

$$\begin{aligned} \beta_1 \approx^{\dagger} \beta_2 &\stackrel{\text{def}}{=} \beta_1 \sqsubseteq^{\dagger} \beta_2 \text{ and } \beta_2 \sqsubseteq^{\dagger} \beta_1 \\ &\Leftrightarrow \forall \mathcal{C}, \forall \mathcal{R} \in \mathcal{C}_{\text{context}}(\mathcal{B}, \mathcal{C}), \forall \vec{t} \in \mathcal{T}_{\text{term}}(\vec{\mathcal{A}}), \left. \begin{array}{l} \mathcal{R}[\beta_1[\vec{t}]] \in \mathcal{R}_{\text{og}} \\ \mathcal{R}[\beta_2[\vec{t}]] \in \mathcal{R}_{\text{og}} \end{array} \right\} \Rightarrow \mathcal{R}[\beta_1[\vec{t}]] \sim \mathcal{R}[\beta_2[\vec{t}]] \end{aligned}$$

While \sqsubseteq and \sqsubset (resp. \approx and \sqsupset) are always preorders (resp. equivalences), \sqsubseteq^{\dagger} (resp. \approx^{\dagger}) may not be because it may not be transitive (as will be explained in [A](#)), which is why we use the word “relation” in place of “preorder” (resp. “equivalence”), and say “symmetric interior”

instead of “induced equivalence”.

Remark C.17

The equivalence

$$p_1 \approx p_2 \Leftrightarrow p_1 \sqsubseteq p_2 \text{ and } p_1 \stackrel{\cdot}{\approx} p_2$$

means that invariants preserved by \approx are exactly those that are preserved by either \sqsubseteq or $\stackrel{\cdot}{\approx}$, which can be thought of as being syntactic and semantic respectively (see the next paragraphs). Similarly, \sqsubseteq and $\stackrel{\cdot}{\approx}$ can be thought of as being the syntactic and semantic parts of \sqsubseteq respectively.

Example C.18

In OCaml^{try}_Z, we have

$$n * n / n \sqsubseteq n : Z.t \approx n + n - n$$

where the inequality being strict is due to $n * n / n$ raising an exception when n is 0, e.g. under $\mathcal{R} = \text{fun } n \rightarrow \square$ on input $i = 0$.

We assert that n is of type $Z.t$ (i.e. use $n : Z.t$ and not n) to account for the fact that it is implicitly asserted in the other two terms by the use of the arithmetic operations. With n , we would instead have

$$n * n / n \sqsubseteq n \sqsupset n + n - n$$

because

$$n : Z.t \sqsubseteq n$$

Indeed, we have:

- $n : Z.t \stackrel{\cdot}{\approx} n$ because types are erased before evaluation, so if both $\mathcal{R} \sqsubseteq n : Z.t$ and $\mathcal{R} \sqsubseteq n$ are programs, they behave the same way; and
- $n : Z.t \sqsubseteq n$ because the type checker can always infer that n has type $Z.t$ when needed; but
- $n : Z.t \not\sqsupset n$ because n may have a type which is incompatible with $Z.t$, e.g. the context

$$\mathcal{R} = \text{fun } m \rightarrow \text{let } n : \text{string} = \text{"a"} \text{ in let } _ = \square \text{ in } 0$$

is such that $\mathcal{R} \sqsubseteq n$ is a program while $\mathcal{R} \sqsubseteq n : Z.t$ is ill-typed and therefore not a program.

Example C.19

In OCaml_Z, we still have

$$n : Z.t \approx n * n / n$$

but $n * n / n$ and $n : Z.t$ are no longer \sqsubseteq -comparable:

$$n * n / n \not\sqsubseteq n : Z.t \quad \text{and} \quad n * n / n \not\sqsupseteq n : Z.t$$

This follows from the possibility of catching the `Division_by_zero` exception and returning a non-zero integers, which allows the two induced programs to return different outputs on input 0. For example, with

$$\mathcal{R} = \text{fun } n \rightarrow \text{try } \square \text{ with Division_by_zero } \rightarrow 1$$

we have

$$\llbracket \mathcal{R} \mid n * n / n \rrbracket(0) = \llbracket \text{fun } n \rightarrow \text{try } n * n / n \text{ with Division_by_zero } \rightarrow 1 \rrbracket(0) = 1$$

and

$$\llbracket \mathcal{R} \mid n : Z.t \rrbracket(0) = \llbracket \text{fun } n \rightarrow \text{try } n : Z.t \text{ with Division_by_zero } \rightarrow 1 \rrbracket(0) = 0$$

so that

$$\mathcal{R} \mid n * n / n \not\sqsubseteq \mathcal{R} \mid n : Z.t \quad \text{and} \quad \mathcal{R} \mid n * n / n \not\sqsupseteq \mathcal{R} \mid n : Z.t$$

Since $\mathcal{R} \mid n * n / n$ and $\mathcal{R} \mid n : Z.t$ are programs, this allows to conclude that

$$n * n / n \not\sqsubseteq n : Z.t \quad \text{and} \quad n * n / n \not\sqsupseteq n : Z.t$$

The observational preorder \sqsubseteq is a preorder which is:

- compositional, i.e. for pairwise compatible fragments $\mathcal{P}_{\ominus}^0 \equiv \mathcal{P}_{\oplus}^0, \dots, \mathcal{P}_{\ominus}^n \equiv \mathcal{P}_{\oplus}^n$ such that $\mathcal{P}_{\ominus}^1, \dots, \mathcal{P}_{\ominus}^n$ are pluggable in \mathcal{P}_{\ominus}^0 and $\mathcal{P}_{\oplus}^1, \dots, \mathcal{P}_{\oplus}^n$ are pluggable in \mathcal{P}_{\oplus}^0 , we have

$$(\forall k \in \{0, \dots, n\}, \mathcal{P}_{\ominus}^k \sqsubseteq \mathcal{P}_{\oplus}^k) \Rightarrow \mathcal{P}_{\ominus}^0 \mid \mathcal{P}_{\ominus}^1, \dots, \mathcal{P}_{\ominus}^n \sqsubseteq \mathcal{P}_{\oplus}^0 \mid \mathcal{P}_{\oplus}^1, \dots, \mathcal{P}_{\oplus}^n$$

In particular, for any contexts $(\mathcal{R}_{\ominus}, \mathcal{R}_{\oplus}) \in \mathcal{C}_{\text{context}}(\mathcal{A}, \mathcal{B})^2$ and terms $(t_{\ominus}, t_{\oplus}) \in \mathcal{T}_{\text{term}}(\mathcal{A})^2$, we have

$$\mathcal{R}_{\ominus} \sqsubseteq \mathcal{R}_{\oplus} \text{ and } t_{\ominus} \sqsubseteq t_{\oplus} \Rightarrow \mathcal{R}_{\ominus} \mid t_{\ominus} \sqsubseteq \mathcal{R}_{\oplus} \mid t_{\oplus}$$

- sound with respect to \lesssim on programs, i.e. for any programs p_1 and p_2 ,

$$p_1 \sqsubseteq p_2 \Rightarrow p_1 \lesssim p_2$$

- program-preserving, i.e. for any terms \mathcal{P}_1 and \mathcal{P}_2 ,

$$\mathcal{P}_1 \sqsubseteq \mathcal{P}_2 \Rightarrow \mathcal{P}_1 \in \mathcal{P}_{\text{rog}} \Rightarrow \mathcal{P}_2 \in \mathcal{P}_{\text{rog}}$$

Similarly, the program-preserving observational equivalence \approx is compositional, sound with respect to \sim on programs, and program-preserving. However, in general, \sqsubseteq (resp. \approx) is not complete with respect to \lesssim (resp. \sim):

Example C.20

Let $\text{OCaml}_{\mathbb{Z}}^{\text{debug}}$ be a variant of $\text{OCaml}_{\mathbb{Z}}$ obtained by replacing the implicit boilerplate code by the code given in Figure C.2 to allow for a `--debug` option whose presence can be checked by a program via the `!debug` boolean^a. We make the `debug` boolean mutable so that debugging can be enabled manually in specific parts of the code if necessary. In this context, the programs

$p_1 = \text{fun } n \rightarrow \text{debug} := \text{true}; n$ and $p_2 = \text{fun } n \rightarrow \text{debug} := \text{false}; n$

represent the same partial functions, i.e. $p_1 \sim p_2$. However, the program-preserving observational equivalence does not equate them, i.e. $p_1 \not\sim p_2$. Indeed, the context

$c = \text{fun } n \rightarrow \text{let } _ = \square \text{ in if !debug then 1 else 2}$

yields programs $c[p_1]$ and $c[p_2]$ such that $c[p_1] \sim c[p_2]$: for any input n ,

$$\llbracket c[p_1] \rrbracket(n) = 1 \neq 2 = \llbracket c[p_2] \rrbracket(n)$$

^aWhere `!` is the dereference operator of OCaml (and not negation).

Figure C.2: The $\text{OCaml}_{\mathbb{Z}}^{\text{debug}}$ boilerplate code

```
let main : Z.t → Z.t =
  let open Z in
   $p$ 
in
let debug = ref false in
let input =
  match Sys.argv with
  | [| _; input_str |] → Z.of_string_base 10 input_str
  | [| _; input_str; "--debug" |]
  | [| _; "--debug"; input_str |] →
    debug := true; Z.of_string_base 10 input_str
  | _ → failwith "Unexpected usage!"
in
let output = main input in
Z.print output
```

Example C.21

In $\text{OCaml}_{\mathbb{Z}}$, this also happens for programs that raise distinct exceptions (that can be caught). For example, the programs

$p_1 = \text{fun } _ \rightarrow 1/0$ and $p_2 = \text{let rec } g \text{ m} = m :: (g (m+1)) \text{ in } g 0$

or equivalently

$p_1 = \text{fun } _ \rightarrow \text{raise Division_by_zero}$ and $p_2 = \text{fun } _ \rightarrow \text{raise Stack_overflow}$ both raise an exception on all inputs, and are hence equivalent as programs, i.e. $p_1 \sim p_2$, because they induce the nowhere-defined function:

$$\llbracket p_1 \rrbracket = \emptyset = \llbracket p_2 \rrbracket$$

However, since they raise distinct exceptions, they are not equated by the program-preserving observational equivalence, i.e. $p_1 \not\sim p_2$. Indeed, the context \mathcal{C} defined as

```
try □ with
| Division_by_zero → 1
| Stack_overflow → 2
```

distinguishes the two kinds of exceptions: for any input n , we have:

$$\llbracket \mathcal{C}[p_1] \rrbracket(n) = 1 \neq 2 = \llbracket \mathcal{C}[p_2] \rrbracket(n)$$

Program preservation: a displeasingly strong syntactic invariant The program-preserving preorder is very syntactic in nature:

Example C.22

In OCaml_ℤ, for simple-enough terms (say those that only use arithmetic operations, functions, if-then-else statements and loops) t_1 and t_2 , we have

$$t_1 \rightsquigarrow t_2 \Leftrightarrow (\forall \Gamma, \forall B, \Gamma \vdash t_1 : B \Rightarrow \Gamma \vdash t_2 : B)$$

where Γ is a typing context and B is a type. Indeed, given $\Gamma = x_1 : A_1, \dots, x_q : A_q$, we can define

$$\mathcal{C} = \text{fun } (x_1 : A_1) \dots (x_q : A_q) \rightarrow (\square : B)$$

and we have

$$\Gamma \vdash t_i : B \Leftrightarrow \mathcal{C}[t_i] \in \mathcal{P}_{\text{rog}}$$

For more complex terms, a similar characterization most likely exists, but Γ and \mathcal{C} may also need to account for declared modules, exceptions etc.

Example C.23

Similarly, in the weak untyped call-by-name / call-by-value λ -calculus, for two λ -terms t_1 and t_2 , we have

$$t_1 \rightsquigarrow t_2 \Leftrightarrow \text{FV}(t_1) \supseteq \text{FV}(t_2)$$

Indeed, given a set of variables $\{x_1, \dots, x_q\}$, we can define

$$\mathcal{C} = \lambda x_1. \dots \lambda x_q. \square$$

and we have

$$\text{FV}(t_i) \subseteq \{x_1, \dots, x_q\} \Leftrightarrow \text{FV}(\mathcal{R}[t_i]) = \emptyset \Leftrightarrow \mathcal{R}[t_i] \in \mathcal{P}_{\text{rog}}$$

This makes $\overline{\sqsubseteq}$ and \approx fairly tedious to work with directly because one has to repeatedly bring up these syntactic invariants:

Example C.24

In OCaml_ℤ, we have

$$0 * n \overline{\sqsubseteq} 0$$

because

$$0 * n \stackrel{\approx}{\sqsubseteq} 0 \quad \text{and} \quad 0 * n \sqsubset 0 \quad \text{but} \quad 0 * n \not\sqsubset 0$$

Indeed, the only difference between the two terms is that $\Gamma \vdash 0 * n : \mathbb{Z}.t$ only holds when $(n : \mathbb{Z}.t) \in \Gamma$ while $\Gamma \vdash 0 : \mathbb{Z}.t$ holds for any Γ .

Example C.25

Similarly, in the weak untyped call-by-name / call-by-value λ -calculus,

$$(\lambda x. I)y \overline{\sqsubseteq} I$$

because

$$(\lambda x. I)y \stackrel{\approx}{\sqsubseteq} I \quad \text{and} \quad (\lambda x. I)y \sqsubset I \quad \text{but} \quad (\lambda x. I)y \not\sqsubset I$$

Indeed, β -conversion \approx_β is sound with respect to $\stackrel{\approx}{\sqsubseteq}$ (i.e. $\approx_\beta \subseteq \stackrel{\approx}{\sqsubseteq}$), and

$$\text{FV}((\lambda x. I)y) = \{y\} \not\supseteq \emptyset = \text{FV}(I)$$

Note that this means that β -conversion \approx_β is not sound with respect to $\overline{\sqsubseteq}$ (though $\approx_\beta \cap \sqsubset$ is).

Weakening syntactic invariants Since we have very simple syntactic characterizations of \sqsubset , to understand the notion of “can be safely replaced” $\overline{\sqsubseteq} = \sqsubset \cap \stackrel{\approx}{\sqsubseteq}$, it suffices to understand $\stackrel{\approx}{\sqsubseteq}$. More generally, for any relation $\stackrel{\mathcal{R}}{\sqsubseteq}$ such that $\overline{\sqsubseteq} \subseteq \stackrel{\mathcal{R}}{\sqsubseteq} \subseteq \stackrel{\approx}{\sqsubseteq}$, we have $\overline{\sqsubseteq} = \sqsubset \cap \stackrel{\mathcal{R}}{\sqsubseteq}$ and it suffices to understand $\stackrel{\mathcal{R}}{\sqsubseteq}$. Such relations are exactly those of the shape $\stackrel{\mathcal{R}}{\sqsubseteq} = \mathcal{R} \cap \stackrel{\approx}{\sqsubseteq}$ for some $\mathcal{R} \supseteq \sqsubset$, i.e. variants of $\overline{\sqsubseteq}$ with \sqsubset weakened to some relation $\mathcal{R} \supseteq \sqsubset$:

Definition C.26

Given a binary relation \mathcal{R} on fragments, we write $\overset{\mathcal{R}}{\sqsubseteq}$ for the intersection $\mathcal{R} \cap \overset{\cdot}{\sqsubseteq}$, i.e.

$$\beta_1 \overset{\mathcal{R}}{\sqsubseteq} \beta_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \beta_1 \mathcal{R} \beta_2, \text{ and} \\ \forall \mathcal{R} \in \mathcal{C}_{\text{context}}, \forall \vec{t} \in \mathcal{T}_{\text{term}}^*, \left(\begin{array}{l} \mathcal{R} \beta_1 \vec{t} \in \mathcal{R}_{\text{rog}} \\ \mathcal{R} \beta_2 \vec{t} \in \mathcal{R}_{\text{rog}} \end{array} \right) \Rightarrow \mathcal{R} \beta_1 \vec{t} \lesssim \mathcal{R} \beta_2 \vec{t} \end{array} \right\}$$

We write $\overset{\mathcal{R}}{\approx}$ for the symmetric interior $\mathcal{R} \cap \mathcal{R}^{-1} \cap \overset{\cdot}{\approx}$ of $\overset{\mathcal{R}}{\sqsubseteq}$, i.e.

$$\beta_1 \overset{\mathcal{R}}{\approx} \beta_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \beta_1 \mathcal{R} \beta_2, \\ \beta_2 \mathcal{R} \beta_1, \text{ and} \\ \forall \mathcal{R} \in \mathcal{C}_{\text{context}}, \forall \vec{t} \in \mathcal{T}_{\text{term}}^*, \left(\begin{array}{l} \mathcal{R} \beta_1 \vec{t} \in \mathcal{R}_{\text{rog}} \\ \mathcal{R} \beta_2 \vec{t} \in \mathcal{R}_{\text{rog}} \end{array} \right) \Rightarrow \mathcal{R} \beta_1 \vec{t} \sim \mathcal{R} \beta_2 \vec{t} \end{array} \right\}$$

Note that when $\mathcal{R} = \varpi$, we get $\overset{\mathcal{R}}{\sqsubseteq} = \overset{\cdot}{\sqsubseteq}$, and when \mathcal{R} is the trivial relation $\mathcal{R} = \mathcal{F}_{\text{rag}}^2$, we get $\overset{\mathcal{R}}{\sqsubseteq} = \overset{\cdot}{\sqsubseteq}$. The question is then: for which relation $\mathcal{R} \supseteq \varpi$ is $\overset{\mathcal{R}}{\sqsubseteq}$ the easiest to study? Using a larger relation \mathcal{R} allows $\overset{\mathcal{R}}{\sqsubseteq}$ to relate more fragments, possibly freeing us from some irrelevant syntactic considerations, but may unfortunately also break transitivity by vacuously relating mismatched fragments (see [A.8](#)). In each calculus, we call our preferred transitive instance of $\overset{\mathcal{R}}{\sqsubseteq}$ (resp. $\overset{\mathcal{R}}{\approx}$) the observational preorder (resp. equivalence), and denote it by \sqsubseteq (resp. \approx).

Example C.27

In the weak untyped call-by-name / call-by-value λ -calculus, we choose $\mathcal{R} = \equiv$, so that

$$\sqsubseteq = \overset{\cdot}{\sqsubseteq} = \equiv \cap \overset{\cdot}{\sqsubseteq} \quad \text{and} \quad \approx = \overset{\cdot}{\approx} = \equiv \cap \overset{\cdot}{\approx}$$

These are transitive because there are context $\lambda x_1 \dots \lambda x_q. \square$ that can close the terms, while preserving and reflecting $\overset{\cdot}{\sqsubseteq}$ (and hence $\overset{\cdot}{\approx}$). Indeed:

- $\overset{\cdot}{\sqsubseteq}$ is transitive on closed terms because all terms are ϖ -equivalent thanks to the fact that

$$\forall \mathbb{K}, \forall T \text{ closed}, \text{FV}(\mathbb{K} \ T) = \text{FV}(\mathbb{K})$$

- this transitivity propagates to all terms because contexts $\lambda x_1 \dots \lambda x_q. \square$ preserve and reflect $\overset{\cdot}{\sqsubseteq}$, i.e.

$$T_1 \overset{\cdot}{\sqsubseteq} T_2 \Leftrightarrow \lambda x_1 \dots \lambda x_q. T_1 \overset{\cdot}{\sqsubseteq} \lambda x_1 \dots \lambda x_q. T_2$$

Given three terms T_1 , T_2 , and T_3 , and \vec{x} an enumeration of their free variables $\text{FV}(T_1) \cup \text{FV}(T_2) \cup \text{FV}(T_3)$, we therefore have

$$T_1 \overset{\cdot}{\sqsubseteq} T_2 \overset{\cdot}{\sqsubseteq} T_3 \Rightarrow \lambda \vec{x}. T_1 \overset{\cdot}{\sqsubseteq} \lambda \vec{x}. T_2 \overset{\cdot}{\sqsubseteq} \lambda \vec{x}. T_3 \Rightarrow \lambda \vec{x}. T_1 \overset{\cdot}{\sqsubseteq} \lambda \vec{x}. T_3 \Rightarrow T_1 \overset{\cdot}{\sqsubseteq} T_3$$



C.3. Distinguishability, separability and binary operational completeness

There are several fairly different ways in which the semantics $\llbracket p_1 \rrbracket$ and $\llbracket p_2 \rrbracket$ of two given programs p_1 and p_2 can differ from one another:

Example C.28

In OCaml_ℤ, we have

$$\begin{aligned} \llbracket \text{fun } n \rightarrow n \rrbracket &= \text{Id}_{\mathbb{Z}} \\ \llbracket \text{fun } n \rightarrow 0 \rrbracket &= 0_{\mathbb{Z}} \\ \llbracket \text{fun } n \rightarrow 1/0 \rrbracket &= \emptyset \\ \llbracket \text{fun } n \rightarrow \text{while true do () done; } 0 \rrbracket &= \emptyset \\ \llbracket \text{fun } n \rightarrow \text{if } n \geq 0 \text{ then } n \text{ else } 1/0 \rrbracket &= \text{Id}_{\mathbb{N}} \\ \llbracket \text{fun } n \rightarrow \text{if } n \leq 0 \text{ then } n \text{ else } 1/0 \rrbracket &= \text{Id}_{-\mathbb{N}} \end{aligned}$$

where \emptyset denotes the nowhere-defined function, $\text{Id}_{\mathbf{X}}$ denotes the identity function on \mathbb{Z} restricted to \mathbf{X} , and $0_{\mathbb{Z}}$ denotes the constant function defined on \mathbb{Z} that always returns 0.

Among the restrictions of $\text{Id}_{\mathbb{Z}}$ (i.e. all the examples above except $0_{\mathbb{Z}}$), some are strictly “better” than others because there is a strict inclusion of their domain of definition, e.g. $\text{Id}_{\mathbb{N}} \subsetneq \text{Id}_{\mathbb{Z}}$, while for others the domains are incomparable (e.g. $\text{Id}_{\mathbb{N}} \not\subseteq \text{Id}_{-\mathbb{N}}$ and $\text{Id}_{-\mathbb{N}} \not\subseteq \text{Id}_{\mathbb{N}}$) but since they are both restrictions of the same function, they can be combined into a function defined on the union of their domain (e.g. $\text{Id}_{\mathbb{N}} \cup \text{Id}_{-\mathbb{N}} = \text{Id}_{\mathbb{N} \cup (-\mathbb{N})} = \text{Id}_{\mathbb{Z}}$). Merging functions in such a way is not possible when they are not restrictions of a same function, i.e. when they return different values on a given input (e.g. $\text{Id}_{\mathbb{N}}(1) = 1 \neq 0 = 0_{\mathbb{Z}}(1)$ so $\text{Id}_{\mathbb{N}} \cup 0_{\mathbb{Z}}$ is not a function).

We therefore define several notions of being different for programs:

Definition C.29

Given two programs p_1 and p_2 , we say that they are:

- not equivalent as programs when they are not \sim -equivalent, i.e. when

$$p_1 \not\sim p_2 \quad \text{or} \quad p_1 \not\approx p_2$$

or equivalently when their interpretations under $\llbracket \cdot \rrbracket$ are not equal, i.e. when

$$\llbracket p_1 \rrbracket \neq \llbracket p_2 \rrbracket$$

- not comparable as programs when they are not \lesssim -comparable, i.e. when

$$p_1 \not\lesssim p_2 \quad \text{and} \quad p_1 \not\gtrsim p_2$$

or equivalently when their interpretations under $\llbracket \cdot \rrbracket$ are not \subseteq -comparable, i.e.

when

$$\llbracket p_1 \rrbracket \not\sqsubseteq \llbracket p_2 \rrbracket \quad \text{and} \quad \llbracket p_1 \rrbracket \not\supseteq \llbracket p_2 \rrbracket$$

- separable as programs when

$$\exists o_1, \exists o_2, \exists i, \llbracket p_1 \rrbracket(i) = o_1 \neq o_2 = \llbracket p_2 \rrbracket(i)$$

or equivalently when their interpretations under $\llbracket \cdot \rrbracket$ are not \subseteq -joinable^a, i.e. when

$$\llbracket p_1 \rrbracket \not\sqsubseteq \llbracket p_2 \rrbracket$$

^aNote that the interpretations of two programs not being \subseteq -joinable (i.e. $\llbracket p_1 \rrbracket \not\sqsubseteq \llbracket p_2 \rrbracket$) is not equivalent to the programs not being \lesssim -joinable (i.e. $p_1 \not\lesssim p_2$): the former asserts that $\llbracket p_1 \rrbracket \cup \llbracket p_2 \rrbracket$ is not a function, while the latter merely asserts that there is no program p_3 such that $\llbracket p_1 \rrbracket \cup \llbracket p_2 \rrbracket = \llbracket p_3 \rrbracket$, which may be strictly weaker for some weird programming languages.

There are clear implications

separable as programs \Rightarrow not comparable as programs \Rightarrow not equivalent as programs
and both implications are often strict.

Example C.30

In OCaml_Z (or OCaml_Z^{try}):

- `fun n → 1/0` and `fun n → while true do () done; 0` are equivalent as programs;
- `fun n → 1/0` and `fun n → n` are not equivalent as programs but are comparable as programs;
- `fun n → if n >= 0 then n else 1/0` and `fun n → if n <= 0 then n else 1/0` are neither comparable as programs nor separable as programs;
- `fun n → n` and `fun n → 0` are separable as programs.

The first two definitions are lifted to fragments in the expected way: by replacing \lesssim by \sqsubseteq . The third definition, separability, is split into two notions: external separability and internal separability. External separability is more or less separability of programs with

$$\llbracket p \rrbracket : \mathcal{I}_{\text{input}} \rightarrow \mathcal{O}_{\text{output}}$$

replaced by

$$\begin{aligned} \llbracket f \rrbracket_{\text{fun}} : \mathcal{F}_{\text{frag}_1} \times \mathcal{F}_{\text{frag}_0}^n \times \mathcal{I}_{\text{input}} &\rightarrow \mathcal{O}_{\text{output}} \\ (\mathbb{R}, \vec{t}, i) &\mapsto \begin{cases} \llbracket \mathbb{R} \boxed{f \vec{t}} \rrbracket(i) & \text{if } \mathbb{R} \boxed{f \vec{t}} \in \mathcal{P}_{\text{rog}} \\ \text{undefined} & \end{cases} \end{aligned}$$

i.e. just like two programs p_1 and p_2 being separable as programs meant that their induced functions $\llbracket p_1 \rrbracket$ and $\llbracket p_2 \rrbracket$ gave different outputs on some input i , two fragments f_1 and f_2

being externally separable means that their induced functions $\llbracket \beta_1 \rrbracket_{\text{fun}}$ and $\llbracket \beta_2 \rrbracket_{\text{fun}}$ give different outputs on some “input” (\mathbb{R}, \vec{t}, i) . Internal separability is a slightly stronger version of external separability, and the distinction between the two will be explained later (see Example C.34).

Definition C.31

Given two fragments β_1 and β_2 , we say that they are:

- not observationally equivalent when they are not \approx -equivalent, i.e. when $\beta_1 \not\sqsubseteq \beta_2$ or $\beta_1 \not\sqsupseteq \beta_2$
- not observationally comparable when they are not \sqsubseteq -comparable, i.e. when $\beta_1 \not\sqsubseteq \beta_2$ and $\beta_1 \not\sqsupseteq \beta_2$
- externally separable when

$\exists \mathbb{R}, \exists \vec{t}$ such that $\mathbb{R} \llbracket \beta_1 \rrbracket \vec{t}$ and $\mathbb{R} \llbracket \beta_2 \rrbracket \vec{t}$ are separable programs

i.e. when

$$\exists \mathbb{R}, \exists \vec{t}, \exists i, \exists o_1, \exists o_2, \left\{ \begin{array}{l} \mathbb{R} \llbracket \beta_1 \rrbracket \vec{t} \in \mathcal{P}_{\text{reg}}, \\ \mathbb{R} \llbracket \beta_2 \rrbracket \vec{t} \in \mathcal{P}_{\text{reg}}, \text{ and} \\ \llbracket \mathbb{R} \llbracket \beta_1 \rrbracket \vec{t} \rrbracket (i) = o_1 \neq o_2 = \llbracket \mathbb{R} \llbracket \beta_2 \rrbracket \vec{t} \rrbracket (i) \end{array} \right.$$

- (internally) separable when

$$\forall p_1, \forall p_2, \exists \mathbb{R}, \exists \vec{t}, \mathbb{R} \llbracket \beta_1 \rrbracket \vec{t} \sim p_1 \text{ and } \mathbb{R} \llbracket \beta_2 \rrbracket \vec{t} \sim p_2$$

There are again clear implications⁶

(internally) separable \Rightarrow externally separable \Rightarrow not obs. comparable \Rightarrow not obs. equivalent

The last two implications are often strict for the same reason as for programs:

⁶The implication

(internally) separable \Rightarrow externally separable

may not always hold under the current assumptions, but this is only because we have made virtually no assumptions on what the programming language can compute. If we were to try and axiomatize a class of programming languages, one of the first axioms we would add is the existence of programs that compute constant functions:

$$\forall o, \exists p, \forall i, \llbracket p \rrbracket (i) = o$$

The implication at hand is an immediate consequence of this axiom: we can just take p_1 and p_2 to be the constant programs that always return o_1 and o_2 respectively.

Example C.32

In $\text{OCaml}_Z^{\text{try}^a}$, the previous examples still have the same relationship as fragments:

- $\text{fun } n \rightarrow 1/0$ and $\text{fun } n \rightarrow \text{while true do () done}; 0$ are observationally equivalent;
- $\text{fun } n \rightarrow 1/0$ and $\text{fun } n \rightarrow n$ are not observationally equivalent, but are observationally comparable;
- $\text{fun } n \rightarrow \text{if } n \geq 0 \text{ then } n \text{ else } 1/0$ and $\text{fun } n \rightarrow \text{if } n \leq 0 \text{ then } n \text{ else } 1/0$ are neither observationally comparable nor externally separable (and hence not internally separable either);
- $\text{fun } n \rightarrow n$ and $\text{fun } n \rightarrow 0$ are (internally) separable.

^aWe use $\text{OCaml}_Z^{\text{try}}$ here because in OCaml_Z , we can use a context to catch the `Division_by_zero` exception, and hence extract information from $1/0$. For example, the context $\mathbb{R} = \text{try } \square \text{ with Division_by_zero} \rightarrow 0$ shows that $\text{fun } n \rightarrow 1/0$ and $\text{fun } n \rightarrow \text{while true do () done}; 0$ are externally separable as fragments, even though they are comparable as programs:

$$\llbracket \mathbb{R} \ p_1 \rrbracket(1) = 1 \neq 0 = \llbracket \mathbb{R} \ p_2 \rrbracket(1)$$

To get examples in OCaml_Z , one can replace $1/0$ by $\text{fun } n \rightarrow \text{while true do () done}; 1$ which can not be observed, even with try-with statements.

The first implication

(internally) separable \Rightarrow externally separable

is the most interesting one in that its non-strictness reflects a sort of internal completeness of the calculus, which we call binary operational completeness:

Definition C.33

A programming language is said to have *binary operational completeness* when any two fragments that are externally separable are (internally) separable, i.e. when external and internal separability coincide.

OCaml_Z has binary operational completeness thanks to the possibility of observing integers:

Example C.34

OCaml_Z has binary operational completeness. Indeed, if we have an input n_0 and outputs $m_1 \neq m_2$ such that

$$\llbracket \mathbb{R} \ t_1 \rrbracket(n_0) = m_1 \text{ and } \llbracket \mathbb{R} \ t_2 \rrbracket(n_0) = m_2$$

then for any programs p_1 and p_2 , we can define

$$\mathbb{R}_2 = \text{fun } n \rightarrow \text{if } \mathbb{R} \ n_0 = m_1 \text{ then } p_1 \text{ n else } p_2 \text{ n}$$

and we have

$$\llbracket \mathbb{R} t_1 \rrbracket = \llbracket p_1 \rrbracket \text{ and } \llbracket \mathbb{R} t_2 \rrbracket = \llbracket p_2 \rrbracket$$

Binary operational completeness can therefore be understood as being more or less about the possibility of observing the outputs internally, i.e. of using them as intermediate results in a larger program. One way to break binary operational completeness is to add a new kind of output which is hidden from the program:

Example C.35

Binary operational completeness can be broken in $\text{OCaml}_{\mathbb{Z}}$ by making $\llbracket \cdot \rrbracket$ record printed strings (on the error output `stderr` since we already use the standard output for the output integer). More precisely, define $\text{OCaml}_{\mathbb{Z}}^{\text{print}}$ as the variant of $\text{OCaml}_{\mathbb{Z}}$ where $\mathcal{O}_{\text{output}}$ is replaced by $\mathcal{O}_{\text{output}} \times \Sigma^*$ (where Σ is the set of printable characters and Σ^* is the set of strings over these characters), where the first component of the product is used as previously, and the second component stores everything that has been printed so far (and is hence initialized to the empty string by `initial_state`). In $\text{OCaml}_{\mathbb{Z}}^{\text{print}}$, we for example have

$$\llbracket \text{fun } n \rightarrow \text{prerr_string "hello"; } n+1 \rrbracket^{\text{print}}(0) = (1, \text{"hello"})$$

The printed string can be observed externally but not from within the $\text{OCaml}_{\mathbb{Z}}^{\text{print}}$ program, so that

$$(\text{internally}) \text{ separable} \not\Leftarrow \text{externally separable}$$

For example, the terms

$$t_1 = \text{prerr_string "a"; } 0 \quad \text{and} \quad t_2 = \text{prerr_string "b"; } 0$$

are externally separable because

$$\llbracket t_1 \rrbracket^{\text{print}}(i) = (0, \text{"a"}) \neq (0, \text{"b"}) = \llbracket t_2 \rrbracket^{\text{print}}(i)$$

but are not (internally) separable^a. Indeed, for any \mathbb{R} , if

$$\llbracket \mathbb{R} t_1 \rrbracket^{\text{print}}(n) = (m_1, s_1) \quad \text{and} \quad \llbracket \mathbb{R} t_2 \rrbracket^{\text{print}}(n) = (m_2, s_2)$$

then we necessarily have $m_1 = m_2$, and we can therefore not have

$$\llbracket \mathbb{R} t_1 \rrbracket^{\text{print}} = \llbracket \text{fun } _ \rightarrow 1 \rrbracket^{\text{print}} \quad \text{and} \quad \llbracket \mathbb{R} t_2 \rrbracket^{\text{print}} = \llbracket \text{fun } _ \rightarrow 2 \rrbracket^{\text{print}}$$

^aNote that this relies on shadowing being disallowed, because otherwise the context could redefine `prerr_string`.

The lack of binary operational completeness can sometimes be fixed by adding something that was “missing” to the language:

Example C.36

In OCaml^{print}_Z, we can try adding a function

```
current_output : unit → string
```

that returns everything that has been printed so far^a. This does allow distinguishing the two terms above thanks to the context

```
 $\mathbb{R} = \text{fun } n \rightarrow \square; \text{if current\_output } () = \text{"a"} \text{ then } P1 \text{ n else } P2 \text{ n}$ 
```

for which we have

```
 $\mathbb{R} \square t_1 \sim^{\text{print}} \text{prerr\_string "a"; } P1$  and  $\mathbb{R} \square t_2 \sim^{\text{print}} \text{prerr\_string "b"; } P2$ 
```

This falls short of showing that the two terms are (internally) separable because the `prerr_string` can not be undone^b (and need to be executed to distinguish the two programs), so that there is no hope of getting

```
 $\mathbb{R}_2 \square t_1 \sim \text{fun } _ \rightarrow 1$  and  $\mathbb{R}_2 \square t_2 \sim \text{fun } _ \rightarrow 2$ 
```

To restore binary operational completeness, we can add yet another function

```
clear_output : unit → string
```

that removes any previously-printed output^c. Its addition makes the two programs (internally) separable thanks to the context

```
 $\text{fun } n \rightarrow \square; \text{let } o = \text{current\_output } () \text{ in clear\_output } (); \text{if } o = \text{"a"} \text{ then } p_1 \text{ n else } p_2 \text{ n}$ 
```

^aThis does not really make sense for `stderr`, but if one were to print to a file instead, this would just amount to reading the file.

^bThis could be understood as revealing a defect of the definition of (internal) separability: it requires a sort of invertibility of the operations. We believe that this interpretation is slightly misguided for several reasons. First, the natural alternative of defining separability of p_1 and p_2 as meaning that

$$\forall p_1, \forall p_2, \exists \mathbb{R}, \mathbb{R} \square t_1 \sim p_1 \text{ and } \mathbb{R} \square t_2 \sim p_2$$

(which uses \sim in place of \sim^{print}) unjustifiably treats `stdout` as being more important than `stderr` (while both play similar roles in $\llbracket \cdot \rrbracket^{\text{print}}$), so that this interpretation may be a consequence of a preference of $\llbracket \cdot \rrbracket$ over $\llbracket \cdot \rrbracket^{\text{print}}$. Secondly, any form of non-invertibility is the consequence of a somewhat arbitrary restriction imposed by the language, the shell, the operating system or something else, so that it suffices to relax this constraint to recover invertibility. For example, while we disallowed variable shadowing, if it were allowed, the contexts \mathbb{R} could redefine `prerr_string` in a way that makes it invertible (e.g. by appending the string to a variable instead of printing it).

^cThis again does not really make sense for printing to `stderr`, but if we were to write to a file, this would simply amount to emptying the file.

C.4. Operational relevance, solvability and unary operational completeness

Internal and external separability are fairly complex notions and it is therefore common to first study the corresponding unary notions first. The naive definition of “unary (internal) separability” of a term t is

$$\forall p, \exists \mathbb{R}, \mathbb{R} \square t \sim p$$

This is a trivial notion in many programming languages. For example, in OCaml_Z, we can always take

$$\mathbb{R} = \text{fun } n \rightarrow \text{let } _ = (\text{fun } n_1 \dots n_q \Rightarrow t) \text{ in } p \ n$$

with n_1, \dots, n_q the free variables of t , and we get

$$\mathbb{R} \boxed{t} \sim p$$

(though we may need to slightly restrict OCaml_Z for this to hold). The intuition is that the term t is simply discarded by \mathbb{R} , so that this is not a real “use” of t : $\llbracket \mathbb{R} \boxed{t} \rrbracket$ is independent of t . This leads to the notion of non-trivial use:

Definition C.37

A *use* of a fragment f is a pair (\mathbb{R}, \vec{t}) such that $\mathbb{R} \boxed{f \vec{t}} \in \mathcal{P}_{\text{rog}}$. A use (\mathbb{R}, \vec{t}) is said to be *non-trivial* when there exists two fragments f_1 and f_2 such that

$$\mathbb{R} \boxed{f_1 \vec{t}} \in \mathcal{P}_{\text{rog}}, \quad \mathbb{R} \boxed{f_2 \vec{t}} \in \mathcal{P}_{\text{rog}}, \quad \text{and} \quad \mathbb{R} \boxed{f_1 \vec{t}} \sim \mathbb{R} \boxed{f_2 \vec{t}}$$

In particular, a use of a term t is a context \mathbb{R} such that $\mathbb{R} \boxed{t} \in \mathcal{P}_{\text{rog}}$, and it is non-trivial when there exists two terms t_1 and t_2 such that

$$\mathbb{R} \boxed{t_1} \in \mathcal{P}_{\text{rog}}, \quad \mathbb{R} \boxed{t_2} \in \mathcal{P}_{\text{rog}}, \quad \text{and} \quad \mathbb{R} \boxed{t_1} \sim \mathbb{R} \boxed{t_2}$$

We can now define the unary versions of external and internal separability, which are called operational relevance and solvability respectively:

Definition C.38

A fragment f is said to be:

- *operationally relevant* when

$$\exists(\mathbb{R}, \vec{t}) \text{ non-trivial use of } f, \exists i, \exists \alpha, \left(\llbracket \mathbb{R} \boxed{f \vec{t}} \rrbracket \right)(i) = \alpha$$

i.e. when

$$\exists(\mathbb{R}, \vec{t}) \text{ non-trivial use of } f, \left(\llbracket \mathbb{R} \boxed{f \vec{t}} \rrbracket \right) \neq \emptyset$$

and *operationally irrelevant* otherwise.

- *solvable* when

$$\forall p, \exists(\mathbb{R}, \vec{t}) \text{ non-trivial use of } f, \mathbb{R} \boxed{f \vec{t}} \sim p$$

and *unsolvable* otherwise.

We have an implication

$$\text{solvable} \Rightarrow \text{operationally relevant}$$

because we can just choose any p such that $\llbracket p \rrbracket \neq \emptyset$ ⁷.

⁷If no such p exists, the implication still holds: non-trivial uses do not exist, so there are no solvable fragments and the implication is vacuously true.

Example C.39

- $\text{fun } n \rightarrow n \text{ and } \text{fun } n \rightarrow 0$ are solvable (and hence operationally relevant) in $\text{OCaml}_{\mathbb{Z}} / \text{OCaml}_{\mathbb{Z}}^{\neg\text{try}}$.
- $\text{fun } n \rightarrow \text{while true do } () \text{ done}; 0$ is not operationally relevant (and hence unsolvable) in $\text{OCaml}_{\mathbb{Z}} / \text{OCaml}_{\mathbb{Z}}^{\neg\text{try}}$.
- $\text{fun } n \rightarrow 1/0$ is solvable (and hence operationally relevant) in $\text{OCaml}_{\mathbb{Z}}^{\neg\text{try}}$, but operationally irrelevant (and hence unsolvable) in $\text{OCaml}_{\mathbb{Z}}$.

Remark C.40

Note that when they exist, operationally irrelevant fragments β_1 are least elements of \sqsubseteq (among fragments that have the same arity), i.e. those such that for any β_2 (of the same arity), we have $\beta_1 \sqsubseteq \beta_2$. Indeed, if $\llbracket \beta_1 \rrbracket \vec{t}$ and $\llbracket \beta_2 \rrbracket \vec{t}$ are both programs, i.e. if $(\llbracket \cdot \rrbracket, \vec{t})$ is a use of β_1 and of β_2 , then either it is a trivial use and we have

$$\llbracket \beta_1 \rrbracket \vec{t} \sim \llbracket \beta_2 \rrbracket \vec{t}$$

or it is a non-trivial use, and by operational irrelevance of β_1 , we get

$$\llbracket \llbracket \beta_1 \rrbracket \vec{t} \rrbracket = \emptyset \subseteq \llbracket \llbracket \beta_2 \rrbracket \vec{t} \rrbracket$$

and hence

$$\llbracket \beta_1 \rrbracket \vec{t} \lesssim \llbracket \beta_2 \rrbracket \vec{t}$$

Remark C.41

In most programming languages, we also have the converse: least elements of \sqsubseteq are operationally irrelevant.

To show that a least element β_1 of \sqsubseteq is operationally irrelevant, we need to show that for any non-trivial use $(\llbracket \cdot \rrbracket, \vec{t})$ of β_1 , we have $\llbracket \llbracket \beta_1 \rrbracket \vec{t} \rrbracket = \emptyset$. At first sight, this looks immediate: we can take any operationally irrelevant fragment β_2 (of the same arity), and since β_1 is a least element, we have $\beta_1 \sqsubseteq \beta_2$, and hence

$$\llbracket \llbracket \beta_1 \rrbracket \vec{t} \rrbracket \subseteq \llbracket \llbracket \beta_2 \rrbracket \vec{t} \rrbracket = \emptyset$$

Unfortunately, this does not work because we have no way of ensuring that $\llbracket \beta_2 \rrbracket \vec{t}$ is a program, i.e. that $(\llbracket \cdot \rrbracket, \vec{t})$ is also a use of β_2 (e.g. β_2 could have free variables that $\llbracket \cdot \rrbracket$ does not bind).

We therefore want to build β_2 in such a way that any use of β_1 is also a use of β_2 , which may not be possible in some programming languages. In most programming

languages, this can be done by using a context \mathbb{R}_0 such that

$$\forall \mathbb{R}, \forall t, \mathbb{R}[t] \in \mathcal{R}_{\text{og}} \Rightarrow \begin{cases} \mathbb{R}[\mathbb{R}_0[t]] \in \mathcal{R}_{\text{og}}, \text{ and} \\ \mathbb{R}_0[t] \text{ is operationally irrelevant} \end{cases}$$

e.g. in OCaml_Z where we can take

$$\mathbb{R}_0 = \text{while true do } (); \square$$

Whenever such a context exists, least element of \sqsubseteq are operationally irrelevant because taking $\mathbb{R}_2 = \mathbb{R}_0[\mathbb{R}_1]$ works.

Just like the equivalence between non-joinability and separability, the equivalence between operational relevance and solvability can be thought of as saying that external results can be used internally, which is why we call it unary operational completeness:

Definition C.42

A programming language is said to have *unary operational completeness* when its operationally relevant fragments are exactly its solvable fragments.

Example C.43

OCaml_Z a priori has unary operational completeness. For terms, this means that any operationally relevant term is solvable. Indeed, let t be an operationally relevant term, we have a non-trivial use \mathbb{R} such that

$$\llbracket \mathbb{R}[t] \rrbracket \neq \emptyset$$

Given an arbitrary program p , we want to find a non-trivial use \mathbb{R}_2 such that

$$\llbracket \mathbb{R}_2[t] \rrbracket = \llbracket p \rrbracket$$

Finding \mathbb{R}_2 such that $\llbracket \mathbb{R}_2[t] \rrbracket = \llbracket p \rrbracket$ is fairly easy, but finding one that is provably a non-trivial use of t is a bit harder. Many cases can be handled simply by using the fact that \mathbb{R} is a non-trivial use, i.e. on the existence of another term u such that $\llbracket \mathbb{R}[t] \rrbracket \neq \llbracket \mathbb{R}[u] \rrbracket$. Indeed, by looking at an input n_0 on which the programs $\mathbb{R}[t]$ and $\mathbb{R}[u]$ disagree (either because they return different outputs, or because one is defined while the other is not), we get three cases:

- If $\llbracket \mathbb{R}[t] \rrbracket(n_0) = m_1 \neq m_2 = \llbracket \mathbb{R}[u] \rrbracket(n_0)$ then taking

$$\mathbb{R}_2 = \text{fun } n \rightarrow \text{if } \mathbb{R} \ n_0 = m_1 \text{ then } p \ n \text{ else } 1 + p \ n$$

works: $\llbracket \mathbb{R}_2[t] \rrbracket = \llbracket p \rrbracket$ and $\llbracket \mathbb{R}_2[u] \rrbracket \neq \llbracket p \rrbracket$ because $\llbracket \mathbb{R}_2[u] \rrbracket(n_0) = 1 + \llbracket p \rrbracket(n_0) \neq \llbracket p \rrbracket(n_0)$

- If $\llbracket \mathbb{R}[t] \rrbracket(n_0)$ is undefined, $\llbracket \mathbb{R}[u] \rrbracket(n_0) = m_0$, and $\llbracket p \rrbracket$ is not total, we can find

n_3 such that $\llbracket p \rrbracket(n_3)$ is undefined and the context

$$\mathcal{C}_2 = \text{fun } n \rightarrow \text{if } n = n_3 \text{ then } \mathcal{C} \ n_0 \text{ else } p \ n$$

works: $\llbracket \mathcal{C}_2 \ t \rrbracket = \llbracket p \rrbracket$ and $\llbracket \mathcal{C}_2 \ u \rrbracket \neq \llbracket p \rrbracket$ because $\llbracket \mathcal{C}_2 \ u \rrbracket(n_3) = \llbracket \mathcal{C}_2 \ u \rrbracket(n_0) = m_0$ while $\llbracket p \rrbracket(n_3)$ is undefined.

- If $\llbracket \mathcal{C} \ t \rrbracket(n_0) = m_0$, $\llbracket \mathcal{C} \ u \rrbracket(n_0)$ is undefined, and $\llbracket p \rrbracket \neq \emptyset$, then we can find n_3 and m_3 such that $\llbracket p \rrbracket(n_3) = m_3$ and the context

$$\mathcal{C}_2 = \text{fun } n \rightarrow \text{if } n = n_3 \text{ then } \mathcal{C} \ n_0 - m_0 + m_3 \text{ else } p \ n$$

works: $\llbracket \mathcal{C}_2 \ t \rrbracket = \llbracket p \rrbracket$ and $\llbracket \mathcal{C}_2 \ u \rrbracket \neq \llbracket p \rrbracket$ because $\llbracket \mathcal{C}_2 \ u \rrbracket(n_3) = \llbracket \mathcal{C}_2 \ u \rrbracket(n_0)$ is undefined while $\llbracket p \rrbracket(n_3) = m_3$.

The remaining cases ($\llbracket \mathcal{C} \ t \rrbracket \subsetneq \llbracket \mathcal{C} \ u \rrbracket$ with $\llbracket p \rrbracket \neq \emptyset$; and $\llbracket \mathcal{C} \ t \rrbracket \supsetneq \llbracket \mathcal{C} \ u \rrbracket$ with $\llbracket p \rrbracket$ total) can a priori not be handled in such a simple way, and require looking more closely at how \mathcal{C} “uses” the terms plugged to replace the context \mathcal{C} and the term u by ones whose shape is known.

Just like binary operational completeness, unary operational completeness can be broken by adding results that can not be observed from within the programming language:

Example C.44

We can extend the notion of output of $\text{OCaml}_{\mathbb{Z}}^{\neg\text{try}}$ by considering exceptions as being outputs, i.e. by defining the set of outputs $\mathcal{O}_{\text{output}}$ as being the union of the set of integers \mathbb{Z} and of the set of (global) exception constructors, with e.g.

$$\llbracket \text{fun } n \rightarrow 1/0 \rrbracket(0) = \llbracket \text{fun } n \rightarrow \text{raise Division_by_zero} \rrbracket(0) = \text{Division_by_zero}$$

This breaks unary operational completeness because $\text{fun } n \rightarrow 1/0$ is operationally relevant (because it returns the output `Division_by_zero` under the trivial context \square on input 0) but not solvable because without try-catch statements, we can never observe exceptions internally.

VII. Call-by-name solvability

VII.1.Reductions and induced notions of evaluation	188
VII.2.Usefulness of the trivial interpretation of programs	202
VII.3.Instanciating the general definitions	203
VII.4.Equivalences between definitions	215



Overview

Three notions of observational equivalence induced by three notions of evaluation
 By varying the programming language structure that we place on the λ -calculus (e.g. the notion of program, of input, of output, of evaluation, ...), one gets many non-equivalent interpretations of programs $\llbracket \cdot \rrbracket$. Somewhat surprisingly, those only induce three non-equivalent notions of observational equivalence \approx (resp. preorder \sqsubseteq) on program fragments, the only relevant parameter being the reduction \triangleright (or more precisely the induced notion of evaluation \triangleright°).

Given a reduction $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \rightarrow\}^1$, we write \approx_\rightsquigarrow for the notion of observational equivalence induced by taking $\triangleright = \rightsquigarrow$. The \rightsquigarrow -observational equivalence \approx_\rightsquigarrow is often defined as testing for \rightsquigarrow -convergence, for β -reducibility to an \rightsquigarrow -normal form, or for β -convertibility to an \rightsquigarrow -normal form, all of which yields equivalent definitions:

$$\begin{aligned} T_N^1 \approx_\rightsquigarrow T_N^2 &\Leftrightarrow \forall \mathbb{K}_N, (\mathbb{K}_N \boxed{T_N^1} \rightsquigarrow^\circ \Leftrightarrow \mathbb{K}_N \boxed{T_N^2} \rightsquigarrow^\circ) \\ &\Leftrightarrow \forall \mathbb{K}_N, (\mathbb{K}_N \boxed{T_N^1} \rightarrow^\circ \rightsquigarrow \Leftrightarrow \mathbb{K}_N \boxed{T_N^2} \rightarrow^\circ \rightsquigarrow) \\ &\Leftrightarrow \forall \mathbb{K}_N, (\mathbb{K}_N \boxed{T_N^1} \approx_{\beta\rightsquigarrow} \Leftrightarrow \mathbb{K}_N \boxed{T_N^2} \approx_{\beta\rightsquigarrow}) \end{aligned}$$

This definition is often simplified by replacing the quantification on arbitrary contexts by a quantification on contexts \mathbb{F}_N of the shape $\mathbb{F}_N = (\lambda x_1^N \dots \lambda x_q^N. \square) T_N^1 \dots T_N^r$. This is (when $q \leq r$) equivalent (up to reductions) to a substitution $\sigma = x_1^N \mapsto T_N^1, \dots, x_q^N \mapsto T_N^q$ and a stack $\mathbb{S}_N = \square T_N^{q+1} \dots T_N^r$, which we like to package into a disubstitution $\varphi = (\sigma, \mathbb{S}_N)$:

$$\begin{aligned} T_N^1 \approx_\rightsquigarrow T_N^2 &\stackrel{\text{def}}{=} \forall \varphi, (T_N^1[\varphi] \rightsquigarrow^\circ \Leftrightarrow T_N^2[\varphi] \rightsquigarrow^\circ) \\ &\Leftrightarrow \forall \sigma, \forall \mathbb{S}_N, (\mathbb{S}_N \boxed{T_N^1[\sigma]} \rightsquigarrow^\circ \Leftrightarrow \mathbb{S}_N \boxed{T_N^2[\sigma]} \rightsquigarrow^\circ) \\ &\Leftrightarrow \forall \mathbb{F}_N, (\mathbb{F}_N \boxed{T_N^1} \rightsquigarrow^\circ \Leftrightarrow \mathbb{F}_N \boxed{T_N^2} \rightsquigarrow^\circ) \end{aligned}$$

The observational equivalence \approx_\triangleright induced by the weak head reduction \triangleright is Abramsky's one [Abr90] (in the so-called lazy λ -calculus); the observational equivalence $\approx_{\overset{h}{\triangleright}}$ induced by the head reduction $\overset{h}{\triangleright}$ is Wadsworth's one [Wad76], and the observational equivalence \approx_\rightarrow induced by the strong reduction \rightarrow is Morris' one [Mor69]². It is well-known that there are strict inclusions [DezGio01]

$$\approx_\triangleright \subsetneq \approx_\rightarrow \subsetneq \approx_{\overset{h}{\triangleright}}$$

(and the strictness of the inclusions can be understood as stemming from a differences of

¹Nearly everything works for many reductions (and sometimes even all reductions), but to avoid discussing extra hypotheses in this summary, we only consider those three reductions.

²Those are sometimes studied via their induced theories, i.e. their restrictions to closed expressions:

$$\mathcal{T}_\rightsquigarrow \stackrel{\text{def}}{=} \approx_\rightsquigarrow \cap (\overline{\mathbf{T}_N} \times \mathbf{T}_N)$$

[Abr90] “The lazy lambda calculus”, Abramsky, 1990


[Wad76] “The Relation Between Computational and Denotational Properties for Scott's D_{infty} -Models of the Lambda-Calculus”, Wadsworth, 1976

[Mor69] “Lambda Calculus Models of Programming Languages”, Morris, 1969

[DezGio01] “From Böhm's Theorem to Observational Equivalences: an Informal Account”, Dezani-Ciancaglini and Giovannetti, 2001

VII. Call-by-name solvability

strength between their respective versions of η -conversion on Böhm trees [IntManPol17]). Relevant references include [Bar84; DezGio01; IntManPol17].

Two notions of operational relevance and one notion of solvability The \rightsquigarrow -operationally irrelevant (resp. \rightsquigarrow -solvable) expressions are defined by: 

$$\begin{aligned} T_N \rightsquigarrow\text{-operationally relevant} &\stackrel{\text{def}}{=} \exists T'_N, \exists \varphi, T_N[\varphi] \rightsquigarrow^* T'_N \not\rightsquigarrow^* \quad (\text{i.e. } \exists \varphi, T_N[\varphi] \rightsquigarrow^{\otimes}) \\ T_N \rightsquigarrow\text{-solvable} &\stackrel{\text{def}}{=} \forall T'_N, \exists \varphi, T_N[\varphi] \rightsquigarrow^* T'_N \end{aligned}$$

These are instances of the general definition, i.e. we can define an interpretation of programs $\llbracket \cdot \rrbracket_{\rightsquigarrow}$ such that those are exactly the expressions that can be used in a program with a non-trivial (resp. an arbitrary) interpretation:

$$\begin{aligned} T_N \rightsquigarrow\text{-operationally relevant} &\Leftrightarrow \exists P \in \mathcal{P}_{\text{rog}, \rightsquigarrow}^{-\emptyset}, \exists \varphi, (\llbracket T_N \rrbracket_{\rightsquigarrow}) = (\llbracket P \rrbracket_{\rightsquigarrow}) \\ T_N \rightsquigarrow\text{-solvable} &\Leftrightarrow \forall P \in \mathcal{P}_{\text{rog}}, \exists \varphi, (\llbracket T_N \rrbracket_{\rightsquigarrow}) = (\llbracket P \rrbracket_{\rightsquigarrow}) \end{aligned}$$

where

$$\mathcal{P}_{\text{rog}, \rightsquigarrow}^{-\emptyset} = \{P \in \mathcal{P}_{\text{rog}} \mid \llbracket P \rrbracket_{\rightsquigarrow} \neq \emptyset\}$$

The observational preorder is defined by

$$T_N^1 \sqsubseteq_{\rightsquigarrow} T_N^2 \stackrel{\text{def}}{=} \forall \varphi, (T_N^1[\varphi] \rightsquigarrow^{\otimes} \Rightarrow T_N^2[\varphi] \rightsquigarrow^{\otimes})$$

and has the \rightsquigarrow -operationally irrelevant expressions as least elements:

$$T_N \rightsquigarrow\text{-operationally irrelevant} \Leftrightarrow \forall U_N, T_N \sqsubseteq_{\rightsquigarrow} U_N$$



Both notions can be characterized in terms of the action of disubstitutions on the set $\mathbf{T}_N / \sim_{\rightsquigarrow}$ expressions modulo \sim_{\rightsquigarrow} : an expression is \rightsquigarrow -operationally relevant (resp. \rightsquigarrow -solvable) exactly when its orbit is non-trivial³ (resp. is all of $\mathbf{T}_N / \sim_{\rightsquigarrow}$). This interpretation in terms of orbits ensures that the strict implications

$$T_N^1 \sim_{\triangleright} T_N^2 \Rightarrow T_N^1 \sim_{\rightarrow} T_N^2 \Rightarrow T_N^1 \sim_{\triangleright} T_N^2$$

induce implications (in the opposite direction) between the different notions of \rightsquigarrow -operational relevance and \rightsquigarrow -solvability. However, some of these induced implications are no longer strict: we only have two non-equivalent notions of operational relevance

$$T_N \text{ is } \triangleright\text{-op. rel.} \Leftarrow T_N \text{ is } \rightarrow\text{-op. rel.} \Leftarrow T_N \text{ is } \overset{h}{\triangleright}\text{-op. rel.}$$

(a counterexample to the \Rightarrow implication being $\lambda x^N. \Omega_N$) and one notion of solvability

$$T_N \text{ is } \triangleright\text{-solvable} \Leftarrow T_N \text{ is } \rightarrow\text{-solvable} \Leftarrow T_N \text{ is } \overset{h}{\triangleright}\text{-solvable}$$

All six of these notions are equivalent to either \triangleright -convergence or $\overset{h}{\triangleright}$ -convergence (and hence have operational characterizations). A summary of the implications between the dif-

³Given an equivalence class $\overline{T_N} \in \mathbf{T}_N / \sim_{\rightsquigarrow}$, its orbit $\{\overline{T_N[\varphi]} \mid \varphi \text{ is a disubstitution}\}$ always contains $\overline{T_N}$ (because we can take φ to be the identity substitution), so by non-trivial orbit, we mean an orbit that contains at least two distinct elements of $\mathbf{T}_N / \sim_{\rightsquigarrow}$.

[IntManPol17] “Refutation of Sallé’s Longstanding Conjecture”, Intrigila, Manzonetto, and Polonsky, 2017

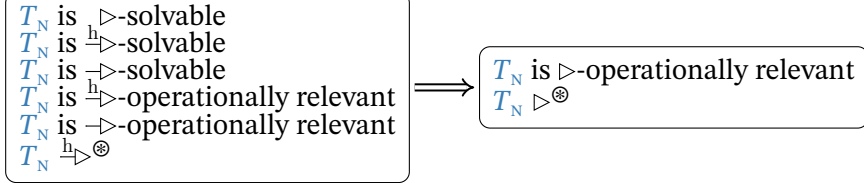
[Bar84] *The lambda calculus: its syntax and semantics*, Barendregt, 1984

[DezGio01] “From Böhm’s Theorem to Observational Equivalences: an Informal Account”, Dezani-Ciancaglini and Giovannetti, 2001

VII. Call-by-name solvability

ferent notions can be found in Figure VII.0.1.

Figure VII.0.1: Implications between notions of \rightsquigarrow -solvability and \rightsquigarrow -operational relevance



Notions in the same node are equivalent, and depicted implications are strict.

Convergence testing as a means to bootstrap meaningful definitions The choice of only testing for convergence is, a priori, unjustified. Indeed, it is natural to first interpret programs P as partial functions

$$\llbracket P \rrbracket : \mathcal{I}_{\text{input}} \rightarrow \mathcal{O}_{\text{output}}$$

and to then define observational equivalence on expressions by

$$T_N \approx U_N \stackrel{\text{def}}{=} \forall \mathbb{K}_N, \begin{cases} \mathbb{K}_N[T_N] \in \mathcal{P}_{\text{og}} \\ \mathbb{K}_N[U_N] \in \mathcal{P}_{\text{og}} \end{cases} \implies \llbracket \mathbb{K}_N[T_N] \rrbracket = \llbracket \mathbb{K}_N[U_N] \rrbracket$$

In the λ -calculus this does not work because there is no natural notion of output. To circumvent this shortfall, the observational equivalence \approx is defined independently of the interpretation of programs $\llbracket \cdot \rrbracket$ by only testing for convergence⁴:

$$T_N \approx_{\rightsquigarrow} U_N \stackrel{\text{def}}{=} \forall \mathbb{K}_N, \begin{cases} \mathbb{K}_N[T_N] \in \mathcal{P}_{\text{og}} \\ \mathbb{K}_N[U_N] \in \mathcal{P}_{\text{og}} \end{cases} \implies \mathbb{K}_N[T_N] \rightsquigarrow^{\otimes} \Leftrightarrow \mathbb{K}_N[U_N] \rightsquigarrow^{\otimes}$$

One can then get a meaningful interpretation of programs $\llbracket \cdot \rrbracket_{\rightsquigarrow}$ by using expressions modulo \approx as outputs⁵:

$$\begin{aligned} \llbracket T_N \rrbracket_{\rightsquigarrow} &: \{\mathbb{S}_N | \mathbb{S}_N \text{ closed}\} \rightarrow \{T_N | T_N \text{ closed and } \rightsquigarrow\text{-normal}\} / \approx_{\rightsquigarrow} \\ \mathbb{S}_N &\mapsto \begin{cases} \overline{U_N} & \text{if } \mathbb{S}_N[T_N] \rightsquigarrow^{\otimes} U_N \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

(where $\overline{U_N}$ is the equivalence class of U_N modulo $\approx_{\rightsquigarrow}$). This turns out to induce the same notion of observational equivalence:

⁴This corresponds to using a meaningless interpretation of programs $\llbracket \cdot \rrbracket_{\rightsquigarrow}$, that uses trivial notions of inputs and outputs. See page ??.

⁵Using non-quotiented expressions as outputs would allow distinguishing more expressions, including some that we a priori do not want to distinguish. See [A.1](#).

VII. Call-by-name solvability

$$T_N \approx_{\rightsquigarrow} U_N \Leftrightarrow \forall \mathbb{K}_N, \begin{matrix} \mathbb{K}_N[T_N] \\ \mathbb{K}_N[U_N] \end{matrix} \in \mathcal{P}_{\text{rog}} \Big\} \Rightarrow \llbracket \mathbb{K}_N[T_N] \rrbracket_{\rightsquigarrow} = \llbracket \mathbb{K}_N[U_N] \rrbracket_{\rightsquigarrow}$$

We therefore end-up having a meaningful interpretation of programs $\llbracket \cdot \rrbracket_{\rightsquigarrow}$ that induces a meaningful notion of observational equivalence $\approx_{\rightsquigarrow}$, even though we had to bootstrap it with some fairly meaningless notions. For $\rightsquigarrow = \multimap$, it is possible to avoid these bootstrapping issues (i.e. it is possible to define $\llbracket \cdot \rrbracket_{\rightsquigarrow}$ before $\approx_{\rightsquigarrow}$) [Mor69] by using the fact that on normal forms, the observational equivalences is exactly $\beta\eta$ -conversion:

$$\{T_N | T_N \text{ closed and } \multimap\text{-normal}\} / \approx_{\multimap} = \{T_N | T_N \text{ closed and } \multimap\text{-normal}\} / \simeq_{\beta\eta}$$

The usefulness of L-calculi Everything above took place in the call-by-name λ -calculus λ_N^{\rightarrow} , and is easily transferred to the intuitionistic call-by-name fragment $\text{Li}_n^{\rightarrow}$ of the $\bar{\lambda}\mu\tilde{\mu}$ -calculus (or its λ -like syntax $\underline{\lambda}_n^{\rightarrow}$), and to its classical call-by-name fragment L_n^{\rightarrow} . All the notions we study are preserved through the inclusion $\text{Li}_n^{\rightarrow} \subseteq L_n^{\rightarrow}$, i.e. adding classical expressions to the calculus has no effect on solvability, operational relevance or the observational preorder⁶.

While the possibility of using disubstitutions in place of contexts is fairly pleasant, using an L-calculus in place of a λ -calculus does not lead to any substantial technical improvements in call-by-name (but will in call-by-value). However, the call-by-name and call-by-value variants of the L-calculi also reveals a few symmetries and asymmetries that were hidden in the corresponding λ -calculi. These will be discussed in Chapter VIII[△].



⁶The intuition is that for these notions, having classical expressions $\mu\alpha^n.c_n$ could only be useful to discard stacks, but since all stacks are of the shape $v_n^1 \cdot \dots \cdot v_n^q \cdot \star^n$, we can already discard them with expressions of the shape $\mu(\underline{}^n \cdot \star^n) \dots \mu(\underline{}^n \cdot \star^n).c_n$.

[Mor69] “Lambda Calculus Models of Programming Languages”, Morris, 1969

VII.1. Reductions and induced notions of evaluation



Five reductions The five reductions we consider in the call-by-name λ -calculus λ_N^\rightarrow are defined in Figure VII.1.1, and the corresponding reductions in L_N^\rightarrow are defined in Figure VII.1.2. The reductions on L_N^\rightarrow induce reductions on Li_N^\rightarrow via restriction, which in turn induce reductions on λ_N^\rightarrow through \Rightarrow . In the λ -calculus literature, three of these reductions are commonly used to model evaluation: the operational (a.k.a. weak head) reduction \triangleright , the head reduction $^h\triangleright$ and the strong reduction \rightarrow . The leftmost-outermost reduction $^{lo}\triangleright$ is also fairly well-known, but is rarely used as the main notion of evaluation. The ahead reduction $^a\triangleright$ is rarely (if ever) used in call-by-name.

As will be shown in Fact VII.1.14, $^h\triangleright$ and $^a\triangleright$ (resp. $^{lo}\triangleright$ and \rightarrow) induce the same notion of evaluation $^h\triangleright^\circledast = ^a\triangleright^\circledast$ (resp. $^{lo}\triangleright^\circledast = \rightarrow^\circledast$). The difference between the two reductions is therefore only that, as depicted in Table VII.1, the former is deterministic but not disubstitutive⁷, while the latter is disubstitutive but not deterministic.

Table VII.1.: Tension between disubstitutivity and determinism in λ_N^\rightarrow , λ_N^\rightarrow , Li_N^\rightarrow , and L_N^\rightarrow

Reduction	\triangleright	$^h\triangleright$	$^a\triangleright$	$^{lo}\triangleright$	\rightarrow
Evaluation	$\triangleright^\circledast$	$^h\triangleright^\circledast = ^a\triangleright^\circledast$	$^h\triangleright^\circledast = ^a\triangleright^\circledast$	$^{lo}\triangleright^\circledast = \rightarrow^\circledast$	$^{lo}\triangleright^\circledast = \rightarrow^\circledast$
Substitutive	✓	✓	✓	✗	✓
Disubstitutive	✓	✗	✓	✗	✓
Deterministic	✓	✓	✗	✓	✗

(See Facts VII.1.4, VII.1.6, VII.1.7, VII.1.8 and VII.1.9)

In most of the call-by-name literature, this tension between determinism and disubstitutivity is resolved in opposite ways for the two notions of evaluation: the deterministic $^h\triangleright$ is used to represent $^h\triangleright^\circledast = ^a\triangleright^\circledast$, while the disubstitutive \rightarrow is used to represent $^{lo}\triangleright^\circledast = \rightarrow^\circledast$. These choices can be justified by disubstitutivity being a much more important than determinism for the proofs to work, and the lack of disubstitutivity of $^h\triangleright$ being easy to work around thanks to its substitutivity and the simple structure of stacks $\mathbb{S}_N = \square U_N^1 \dots U_N^q$. Here, we nevertheless use all five reductions (\triangleright , $^h\triangleright$, $^a\triangleright$, $^{lo}\triangleright$, and \rightarrow), for two reasons. First, doing so allows for more uniform proofs that work for both \triangleright and the other reductions: for each proposition, we can choose to require either determinism (i.e. to use \triangleright , $^h\triangleright$, or $^{lo}\triangleright$), or disubstitutivity (i.e. to use \triangleright , $^a\triangleright$, or \rightarrow). Secondly, while using $^a\triangleright$ is somewhat superfluous in call-by-name, it will become important in call-by-value because the call-by-value variant of $^h\triangleright$ will no longer be substitutive.

⁷Recall that in λ_N^\rightarrow , we call disubstitutions φ pairs (σ, \mathbb{S}_N) where σ is a substitution and \mathbb{S}_N is a stack, that we write $T_N[\varphi]$ for $\mathbb{S}_N[T_N[\sigma]]$; and that a reduction \rightsquigarrow is said to be disubstitutive when

$$T_N \rightsquigarrow T'_N \Rightarrow \forall \varphi, T_N[\varphi] \rightsquigarrow T'_N[\varphi] \quad (\text{i.e. } \forall \sigma, \forall \mathbb{S}_N, \mathbb{S}_N[T_N[\sigma]] \rightsquigarrow \mathbb{S}_N[T'_N[\sigma]])$$

VII. Call-by-name solvability

Figure VII.1.1: Reductions in λ_N^{\rightarrow}

Figure VII.1.1.a: Top-level reduction

$$\frac{}{\text{let } x^N := T_N \text{ in } U_N \triangleright U_N[T_N/x^N]} R_{\text{let}}^T \quad \frac{}{(\lambda x^N. T_N) U_N \triangleright T_N[U_N/x^N]} R_{\rightarrow}^T$$

Figure VII.1.1.b: Operational reduction (a.k.a. weak head reduction)

$$\frac{T_N \triangleright T'_N}{T_N \triangleright T'_N} R_{\triangleright}^O \quad \frac{T_N \triangleright T'_N}{T_N U_N \triangleright T'_N U_N} R_{@_l}^O$$

Figure VII.1.1.c: Leftmost outermost ahead reduction (a.k.a head reduction)

$$\frac{T_N \triangleright T'_N}{T_N \text{ h}\triangleright T'_N} R_{\triangleright}^H \quad \frac{T_N \text{ h}\triangleright T'_N}{\lambda x^N. T_N \text{ h}\triangleright \lambda x^N. T'_N} R_{\lambda}^H$$

Figure VII.1.1.d: Ahead reduction

$$\frac{T_N \triangleright T'_N}{T_N \text{ a}\triangleright T'_N} R_{\triangleright}^A \quad \frac{T_N \text{ a}\triangleright T'_N}{\lambda x^N. T_N \text{ a}\triangleright \lambda x^N. T'_N} R_{\lambda}^A$$

$$\frac{T_N \text{ a}\triangleright T'_N}{T_N U_N \text{ a}\triangleright T'_N U_N} R_{@_l}^A \quad \frac{U_N \text{ a}\triangleright U'_N}{\text{let } x^N := T_N \text{ in } U_N \text{ a}\triangleright \text{let } x^N := T_N \text{ in } U'_N} R_{\text{let}_r}^A$$

Figure VII.1.1.e: Leftmost outermost reduction

$$\frac{T_N \triangleright T'_N}{T_N \text{ lo}\triangleright T'_N} R_{\triangleright}^{\text{LO}} \quad \frac{T_N \text{ lo}\triangleright T'_N}{\lambda x^N. T_N \text{ lo}\triangleright \lambda x^N. T'_N} R_{\lambda}^{\text{LO}}$$

$$\frac{U_N \text{ lo}\triangleright U'_N}{x^N T_N^{\text{NF},1} \dots T_N^{\text{NF},r-1} U_N T_N^{r+1} \dots T_N^q \text{ lo}\triangleright x^N T_N^{\text{NF},1} \dots T_N^{\text{NF},r-1} U'_N T_N^{r+1} \dots T_N^q} R_{@}^{\text{LO}}$$

(where each $T_N^{\text{NF},k}$ ranges over the normal forms described in Figure VII.1.3)

VII. Call-by-name solvability

Figure VII.1.1.f: Strong reduction

$$\begin{array}{c}
\frac{T_N \triangleright T'_N}{T_N \rightarrow T'_N} R_{\triangleright}^S \quad \frac{T_N \rightarrow T'_N}{\lambda x^N. T_N \rightarrow \lambda x^N. T'_N} R_{\lambda}^S \\
\\
\frac{T_N \rightarrow T'_N}{T_N U_N \rightarrow T'_N U_N} R_{@_l}^S \quad \frac{U_N \rightarrow U'_N}{\text{let } x^N := T_N \text{ in } U_N \rightarrow \text{let } x^N := T'_N \text{ in } U'_N} R_{\text{let}_r}^S \\
\\
\frac{U_N \rightarrow U'_N}{T_N U_N \rightarrow T_N U'_N} R_{@_r}^S \quad \frac{T_N \rightarrow T'_N}{\text{let } x^N := T_N \text{ in } U_N \rightarrow \text{let } x^N := T'_N \text{ in } U_N} R_{\text{let}_l}^S
\end{array}$$

Figure VII.1.2: Reductions in L_n^-

Figure VII.1.2.a: Operational reduction / top-level reduction

$$\begin{array}{c}
\frac{}{\langle \mu \alpha^n. c_n | s_n \rangle \triangleright s_n[s_n / \alpha^n]} R_{\mu}^O \quad \frac{}{\langle t_n | \tilde{\mu} x^n. c_n \rangle \triangleright c_n[t_n / x^n]} R_{\tilde{\mu}}^O \\
\\
\frac{}{\langle \mu(x^n \cdot \alpha^n). c_n | v_n \cdot s_n \rangle \triangleright c_n[v_n / x^n, s_n / \alpha^n]} R_{\rightarrow}^O
\end{array}$$

Figure VII.1.2.b: Leftmost outermost ahead reduction (a.k.a head reduction)

$$\begin{array}{c}
\frac{c_n \triangleright c'_n}{c_n \multimap c'_n} R_{\triangleright}^H \quad \frac{\mu(x^n \cdot \beta^n). c_n \multimap \mu(x^n \cdot \beta^n). c'_n}{\langle \mu(x^n \cdot \beta^n). c_n | \alpha^n \rangle \multimap \langle \mu(x^n \cdot \beta^n). c_n | \alpha^n \rangle} R_{\langle \cdot | \alpha \rangle}^H \\
\\
\frac{c_n \multimap c'_n}{\mu \alpha^n. c_n \multimap \mu \alpha^n. c'_n} R_{\mu \alpha}^H \quad \frac{c_n \multimap c'_n}{\tilde{\mu} x^n. c \multimap \tilde{\mu} x^n. c'} R_{\tilde{\mu} x}^H \\
\\
\frac{c_n \multimap c'_n}{\mu(x^n \cdot \alpha^n). c_n \multimap \mu(x^n \cdot \alpha^n). c'_n} R_{\mu \rightarrow}^H
\end{array}$$

VII. Call-by-name solvability

Figure VII.1.2.c: Ahead reduction

$$\begin{array}{c}
\frac{c_n \triangleright c'_n}{c_n \dot{\triangleright} c'_n} R^A \quad \frac{t_n \dot{\triangleright} t'_n}{\langle t_n | s_n \rangle \dot{\triangleright} \langle t'_n | s_n \rangle} R^A_{\langle \cdot | s \rangle} \quad \frac{e_n \dot{\triangleright} e'_n}{\langle t_n | e_n \rangle \dot{\triangleright} \langle t_n | e'_n \rangle} R^A_{\langle t | \cdot \rangle} \\
\\
\frac{c_n \dot{\triangleright} c'_n}{\mu \alpha^n . c_n \dot{\triangleright} \mu \alpha^n . c'_n} R^A_{\mu \alpha} \quad \frac{c_n \dot{\triangleright} c'_n}{\tilde{\mu} x^n . c \dot{\triangleright} \tilde{\mu} x^n . c'} R^A_{\tilde{\mu} x} \\
\\
\frac{c_n \dot{\triangleright} c'_n}{\mu(x^n \cdot \alpha^n) . c_n \dot{\triangleright} \mu(x^n \cdot \alpha^n) . c'_n} R^A_{\mu \rightarrow}
\end{array}$$

Figure VII.1.2.d: Leftmost outermost reduction

$$\begin{array}{c}
\frac{c_n \triangleright c'_n}{c_n \dot{\triangleright} c'_n} R^{LO} \quad \frac{\mu(x^n \cdot \beta^n) . c_n \dot{\triangleright} \mu(x^n \cdot \beta^n) . c'_n}{\langle \mu(x^n \cdot \beta^n) . c_n | \alpha^n \rangle \dot{\triangleright} \langle \mu(x^n \cdot \beta^n) . c'_n | \alpha^n \rangle} R^{LO}_{\langle \cdot | \alpha \rangle} \\
\\
\frac{s_n \dot{\triangleright} s'_n}{\langle x^n | s_n \rangle \dot{\triangleright} \langle x^n | s'_n \rangle} R^{LO}_{\langle x | \cdot \rangle} \quad \frac{c_n \dot{\triangleright} c'_n}{\mu \alpha^n . c_n \dot{\triangleright} \mu \alpha^n . c'_n} R^{LO}_{\mu \alpha} \quad \frac{c_n \dot{\triangleright} c'_n}{\tilde{\mu} x^n . c \dot{\triangleright} \tilde{\mu} x^n . c'} R^{LO}_{\tilde{\mu} x} \\
\\
\frac{c_n \dot{\triangleright} c'_n}{\mu(x^n \cdot \alpha^n) . c_n \dot{\triangleright} \mu(x^n \cdot \alpha^n) . c'_n} R^{LO}_{\mu \rightarrow} \quad \frac{v_n \dot{\triangleright} v'_n}{v_n \cdot s_n \dot{\triangleright} v'_n \cdot s_n} R^{LO}_{\rightarrow, v} \\
\\
\frac{s_n \dot{\triangleright} s'_n}{v_n^{NF} \cdot s_n \dot{\triangleright} v_n^{NF} \cdot s'_n} R^{LO}_{\rightarrow, s}
\end{array}$$

(where each v_n^{NF} ranges over the normal forms described in Figure VII.1.3)

Figure VII.1.2.e: Strong reduction

$$\begin{array}{c}
\frac{c_n \triangleright c'_n}{c_n \dot{\triangleright} c'_n} R^S \quad \frac{t_n \dot{\triangleright} t'_n}{\langle t_n | e_n \rangle \dot{\triangleright} \langle t'_n | e_n \rangle} R^S_{\langle \cdot | e \rangle} \quad \frac{e_n \dot{\triangleright} e'_n}{\langle t_n | e_n \rangle \dot{\triangleright} \langle t_n | e'_n \rangle} R^S_{\langle t | \cdot \rangle} \\
\\
\frac{c_n \dot{\triangleright} c'_n}{\mu \alpha^n . c_n \dot{\triangleright} \mu \alpha^n . c'_n} R^S_{\mu \alpha} \quad \frac{c_n \dot{\triangleright} c'_n}{\tilde{\mu} x^n . c \dot{\triangleright} \tilde{\mu} x^n . c'} R^S_{\tilde{\mu} x} \quad \frac{v_n \dot{\triangleright} v'_n}{v_n \cdot s_n \dot{\triangleright} v'_n \cdot s_n} R^S_{\rightarrow, v} \\
\\
\frac{s_n \dot{\triangleright} s'_n}{v_n \cdot s_n \dot{\triangleright} v_n \cdot s'_n} R^S_{\rightarrow, s} \quad \frac{c_n \dot{\triangleright} c'_n}{\mu(x^n \cdot \alpha^n) . c_n \dot{\triangleright} \mu(x^n \cdot \alpha^n) . c'_n} R^S_{\mu \rightarrow}
\end{array}$$

VII. Call-by-name solvability

These reductions form a lattice under inclusion:

Fact VII.1.1

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , the following inclusions hold:

$$\begin{array}{ccccc} \triangleright & \subseteq & \overset{h}{\triangleright} & \subseteq & \overset{lo}{\triangleright} \\ & & \cap & & \cap \\ & & \overset{a}{\triangleright} & \subseteq & \rightarrow \end{array}$$

Proof

By induction on the derivations.

Normal forms The different kind of normal forms are described in Figure VII.1.3⁸. The normal forms in Li_n^\rightarrow are described with the same grammar are those of L_n^\rightarrow with every stack variable replaced by \star^n , i.e. the sets of normal forms of Li_n^\rightarrow are exactly the intersection of Li_n^\rightarrow with the corresponding set of normal forms in L_n^\rightarrow .

Fact VII.1.2: Shape of normal forms

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , the operational (resp. ahead, strong) normal forms described in Figure VII.1.3 are exactly the \triangleright -normal (resp. $\overset{a}{\triangleright}$ -normal, \rightarrow -normal) forms, the $\overset{h}{\triangleright}$ -normal forms are exactly the $\overset{a}{\triangleright}$ -normal forms, and the $\overset{lo}{\triangleright}$ -normal forms are exactly the \rightarrow -normal forms:

$$\begin{array}{llll} \text{Operational normal form} & \Leftrightarrow & \triangleright\text{-normal} & \\ \text{Ahead normal form} & \Leftrightarrow & \overset{a}{\triangleright}\text{-normal} & \Leftrightarrow \overset{h}{\triangleright}\text{-normal} \\ \text{(Strong) normal form} & \Leftrightarrow & \rightarrow\text{-normal} & \Leftrightarrow \overset{lo}{\triangleright}\text{-normal} \end{array}$$

Proof

Each implication is proven by induction on the syntax, by induction on the derivation, or by the inclusions of Fact VII.1.1.

This implies that $\overset{h}{\triangleright}$ (resp. $\overset{lo}{\triangleright}$) is a (deterministic) strategy for $\overset{a}{\triangleright}$ (resp. \rightarrow):

Fact VII.1.3

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , the reduction $\overset{h}{\triangleright}$ (resp. $\overset{lo}{\triangleright}$) is a strategy for $\overset{a}{\triangleright}$ (resp. \rightarrow):

$$\overset{h}{\triangleright} \subseteq \overset{a}{\triangleright} \quad \text{and} \quad NF_{\overset{h}{\triangleright}} = NF_{\overset{a}{\triangleright}} \quad (\text{resp. } \overset{lo}{\triangleright} \subseteq \rightarrow \quad \text{and} \quad NF_{\overset{lo}{\triangleright}} = NF_{\rightarrow})$$

⁸We denote the $\overset{a}{\triangleright}$ -normal (or equivalently $\overset{h}{\triangleright}$ -normal) expressions by T_N^{AHNF} because we think of $\overset{h}{\triangleright}$ as being just a deterministic strategy for $\overset{a}{\triangleright}$ (and we do not denote them by T_N^{ANF} because ANF is often used to abbreviate A-normal form, an a priori unrelated notion).

VII. Call-by-name solvability

Proof

Immediate by Facts VII.1.1 and VII.1.2.

Substitutivity and disubstitutivity Four of the reductions are substitutive, and three of those are also closed under stacks (and hence disubstitutive):

Fact VII.1.4: Substitutivity of \triangleright , $\overset{h}{\triangleright}$, $\overset{a}{\triangleright}$ and $\rightarrow\triangleright$ in λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , each reduction $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{a}{\triangleright}, \rightarrow\triangleright\}$ is substitutive: for any terms t and t' and any substitution σ ,

$$t \rightsquigarrow t' \Rightarrow t[\sigma] \rightsquigarrow t'[\sigma]$$

Proof

By induction on the derivation, and substitutivity of \triangleright or \triangleright .

Fact VII.1.5: Non substitutivity of $\overset{lo}{\triangleright}$ in λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , the leftmost-outermost reduction $\overset{lo}{\triangleright}$ is not substitutive.

Proof

Immediate:

- In λ_N^\rightarrow , we have

$$x^N(I_N y^N) \overset{lo}{\triangleright} x^N y^N$$

but this reduction is not preserved through the substitution $x^N \mapsto \lambda_{-^N}.I_N$:

$$(\lambda_{-^N}.I_N)(I_N y^N) \overset{lo}{\triangleright} I_N \neq (\lambda_{-^N}.I_N) y^N$$

- In L_n^\rightarrow , we have

$$\langle x^n | (\mu \beta^n. \langle I_n | y^n \cdot \beta^n \rangle) \cdot \alpha^n \rangle \overset{lo}{\triangleright} \langle x^n | (\mu \beta^n. \langle y^n | \beta^n \rangle) \cdot \alpha^n \rangle$$

but this reduction is not preserved through the substitution $x^N \mapsto \mu(-^n \cdot \gamma^n). \langle I_n | \gamma^n \rangle$:

$$\langle \mu(-^n \cdot \gamma^n). \langle I_n | \gamma^n \rangle | (\mu \beta^n. \langle I_n | y^n \cdot \beta^n \rangle) \cdot \alpha^n \rangle \overset{lo}{\triangleright} \langle I_n | \gamma^n \rangle \neq \langle \mu(-^n \cdot \gamma^n). \langle I_n | \gamma^n \rangle | (\mu \beta^n. \langle y^n | \beta^n \rangle) \cdot \alpha^n \rangle$$

VII. Call-by-name solvability

Fact VII.1.6: Disubstitutivity of \triangleright , $\overset{a}{\triangleright}$ and $\neg\triangleright$ in λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , each reduction $\rightsquigarrow \in \{\triangleright, \overset{a}{\triangleright}, \neg\triangleright\}$ is disubstitutive: for any terms t and t' and any disubstitution φ ,

$$t \rightsquigarrow t' \Rightarrow t[\varphi] \rightsquigarrow t'[\varphi]$$

In particular:

- In λ_N^\rightarrow , they are closed under stacks \mathbb{S}_N :

$$T_N \rightsquigarrow T'_N \Rightarrow \mathbb{S}_N[T_N] \rightsquigarrow \mathbb{S}_N[T'_N]$$

- In λ_n^\rightarrow , they are closed under defer:

$$o \rightsquigarrow o' \Rightarrow \text{defer}(o, s_n) \rightsquigarrow \text{defer}(o', s_n)$$

- In Li_n^\rightarrow , they are closed under substitution of the stack variable \star^n by a stack s_n :

$$o \rightsquigarrow o' \Rightarrow o[s_n/\star^n] \rightsquigarrow o'[s_n/\star^n]$$

Proof

Closure under stacks in λ_N^\rightarrow is by induction on \mathbb{S}_N . Disubstitutivity in λ_N^\rightarrow follows from closure under stacks and substitutivity (Fact VII.1.4). Disubstitutivity in L_n^\rightarrow (and hence in Li_n^\rightarrow) is by induction on the derivation and disubstitutivity of \triangleright (Fact ??).

Fact VII.1.7: Non-disubstitutivity of $\overset{h}{\triangleright}$ and $\overset{lo}{\triangleright}$ in λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , the reductions $\overset{h}{\triangleright}$ and $\overset{lo}{\triangleright}$ are not disubstitutive. Furthermore:

- In λ_N^\rightarrow , they are not closed under stacks;
- In λ_n^\rightarrow , they are not closed under defer;
- In Li_n^\rightarrow (resp. L_n^\rightarrow), they are not closed under disubstitution of the shape $\star^n \mapsto s_n$ (resp. $\alpha^n \mapsto s_n$).

Proof

- In λ_N^\rightarrow , the reductions $\rightsquigarrow \in \{\overset{h}{\triangleright}, \overset{lo}{\triangleright}\}$ are not closed under the stack $\square y^N$. Indeed, we have

$$\lambda_{-}^N.I_N x^N \rightsquigarrow \lambda_{-}^N.x^N$$

but

$$I_N x^N \leftarrow (\lambda_{-}^N.I_N x^N)y^N \not\rightsquigarrow (\lambda_{-}^N.x^N)y^N$$

VII. Call-by-name solvability

- In Li_n^\rightarrow , the reductions $\rightsquigarrow \in \{\overset{h}{\triangleright}, \overset{lo}{\triangleright}\}$ are not closed under the disubstitution $\star^n \mapsto y^n \cdot \star^n$. Indeed, we have

$$\langle \mu(_{}^n \cdot \star^n). \langle I_n | x^n \cdot \star^n \rangle | \star^n \rangle \rightsquigarrow \langle \mu(_{}^n \cdot \star^n). \langle x^n | \star^n \rangle | \star^n \rangle$$

but

$$\langle I_n | x^n \cdot \star^n \rangle \not\rightsquigarrow \langle \mu(_{}^n \cdot \star^n). \langle I_n | x^n \cdot \star^n \rangle | y^n \cdot \star^n \rangle \not\rightsquigarrow \langle \mu(_{}^n \cdot \star^n). \langle x^n | \star^n \rangle | \star^n \rangle$$

Determinism, confluence, and uniqueness of termination behavior All five reductions have some form of determinism, ranging from actual determinism to confluence:

Fact VII.1.8: Determinism of \triangleright , $\overset{h}{\triangleright}$ and $\overset{lo}{\triangleright}$ in λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow , each reduction $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{lo}{\triangleright}\}$ is deterministic: for any terms t , t_\ominus , and t_\oplus ,

$$t_\ominus \leftarrow t \rightsquigarrow t_\oplus \Rightarrow t_\ominus = t_\oplus$$

Proof

By induction on the derivation.

Fact VII.1.9: Non-determinism of $\overset{a}{\triangleright}$ and \rightarrow in λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow , the reductions $\overset{a}{\triangleright}$ and \rightarrow are non-deterministic.

Proof

Since $\overset{a}{\triangleright} \subseteq \rightarrow$, it suffices to show that $\overset{a}{\triangleright}$ is not deterministic.

- In λ_N^\rightarrow , the reduction $\overset{a}{\triangleright}$ is not deterministic because

$$I_N I_N \triangleleft^{\overset{a}{\triangleright}} (\lambda_{}^N. I_N I_N) x^N \overset{a}{\triangleright} (\lambda_{}^N. I_N) x^N$$

(and $I_N I_N \neq (\lambda_{}^N. I_N) x^N$).

- In Li_n^\rightarrow , the reduction $\overset{a}{\triangleright}$ is not deterministic because

$$\langle I_n | I_n \cdot \star^n \rangle \triangleleft^{\overset{a}{\triangleright}} \langle \mu(_{}^n \cdot \star^n). \langle I_n | I_n \cdot \star^n \rangle | x^n \cdot \star^n \rangle \overset{a}{\triangleright} \langle \mu(_{}^n \cdot \star^n). \langle I_n | \star^n \rangle | x^n \cdot \star^n \rangle$$

(and $\langle I_n | I_n \cdot \star^n \rangle \neq \langle \mu(_{}^n \cdot \star^n). \langle I_n | \star^n \rangle | x^n \cdot \star^n \rangle$).

VII. Call-by-name solvability

Fact VII.1.10: Uniform confluence of $\overset{a}{\triangleright}$ in λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow , each reduction $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{a}{\triangleright}, \overset{lo}{\triangleright}\}$ is uniformly confluent: for any terms t , t_\ominus , and t_\oplus ,

$$t_\ominus \leftarrow t \rightsquigarrow t_\oplus \Rightarrow t_\ominus = t_\oplus \text{ or } t_\ominus \rightsquigarrow \leftarrow t_\oplus$$

Proof

For $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{lo}{\triangleright}\}$, this immediately follows from \rightsquigarrow being deterministic. For $\rightsquigarrow = \overset{a}{\triangleright}$, this is proven by induction on the derivations.

Fact VII.1.11: Confluence of $\neg\triangleright$ in λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow

In λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow , each reduction $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{a}{\triangleright}, \overset{lo}{\triangleright}, \neg\triangleright\}$ is confluent: for any terms t , t_\ominus , and t_\oplus ,

$$t_\ominus \leftarrow^* t \rightsquigarrow^* t_\oplus \Rightarrow t_\ominus \rightsquigarrow \leftarrow^* t_\oplus$$

Proof

For $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{a}{\triangleright}, \overset{lo}{\triangleright}\}$, this follows from \rightsquigarrow being uniformly confluent (see [A.1](#)). For $\rightsquigarrow = \neg\triangleright$, this is easily proven using the parallel reduction: for λ_N^\rightarrow , see Definition 3.2.3 and Lemma 3.2.4 page 60 of [Bar84], and for L_n^\rightarrow , see Proposition ??.

Fact VII.1.12

In λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow , each reduction $\rightsquigarrow \in \{\triangleright, \overset{h}{\triangleright}, \overset{lo}{\triangleright}, \overset{a}{\triangleright}\}$ has uniqueness of termination behavior: for any term t ,

$$t \rightsquigarrow^\circledast \Leftrightarrow \neg(t \rightsquigarrow^\omega)$$

Proof

This follows from these reductions being uniformly confluent (see [A.1](#)).

Three notions of evaluation Given a reduction \rightsquigarrow , the induced notion of evaluation $\rightsquigarrow^\circledast$ is defined by

$$t \rightsquigarrow^\circledast t' \stackrel{\text{def}}{=} t \rightsquigarrow^* t' \text{ and } \neg \exists t'', t' \rightsquigarrow t''$$

Note that having $\rightsquigarrow_1 \subseteq \rightsquigarrow_2$ implies that $\rightsquigarrow_1^* \subseteq \rightsquigarrow_2^*$ (i.e. $\rightsquigarrow \mapsto \rightsquigarrow^*$ is positively monotone) and $\rightsquigarrow_1 \supseteq \rightsquigarrow_2$ (i.e. $\rightsquigarrow \mapsto \rightsquigarrow^\circledast$ is negatively monotone), but implies neither $\rightsquigarrow_1^\circledast \subseteq \rightsquigarrow_2^\circledast$ nor $\rightsquigarrow_1^\circledast \supseteq \rightsquigarrow_2^\circledast$ (i.e. $\rightsquigarrow \mapsto \rightsquigarrow^\circledast$ is not monotone). There are therefore 5, a priori distinct, notions

VII. Call-by-name solvability

of evaluation: \triangleright^\otimes , $\overset{h}{\triangleright}^\otimes$, $\overset{a}{\triangleright}^\otimes$, $\overset{lo}{\triangleright}^\otimes$ and $\neg\triangleright^\otimes$. By using the fact that $\overset{h}{\triangleright}$ (resp. $\overset{lo}{\triangleright}$) is a strategy for $\overset{a}{\triangleright}$ (resp. $\neg\triangleright$), we nevertheless get inclusions:

Fact VII.1.13

If $\rightsquigarrow_\diamond$ is a strategy of \rightsquigarrow , then $\rightsquigarrow_\diamond^\otimes \subseteq \rightsquigarrow^\otimes$:

$$\left. \begin{array}{c} \rightsquigarrow_\diamond \subseteq \rightsquigarrow \\ \forall t, t \rightsquigarrow_\diamond \Leftrightarrow t \rightsquigarrow \end{array} \right\} \Rightarrow \rightsquigarrow_\diamond^\otimes \subseteq \rightsquigarrow^\otimes$$

In particular, we have

$$\overset{h}{\triangleright}^\otimes \subseteq \overset{a}{\triangleright}^\otimes \quad \text{and} \quad \overset{lo}{\triangleright}^\otimes \subseteq \neg\triangleright^\otimes$$

Proof

Immediate.

By using factorization properties, we can then conclude that there are only three notions of evaluation:

Fact VII.1.14

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , we have

$$\overset{h}{\triangleright}^\otimes = \overset{a}{\triangleright}^\otimes \quad \text{and} \quad \overset{lo}{\triangleright}^\otimes = \neg\triangleright^\otimes$$

Proof sketch ([See page 233 for details](#))

Both \subseteq inclusions follow from the previous fact. The inclusion $\overset{h}{\triangleright}^\otimes \supseteq \overset{a}{\triangleright}^\otimes$ follows from $\overset{a}{\triangleright}$ having uniqueness of termination behavior, and the inclusion $\overset{lo}{\triangleright}^\otimes \supseteq \neg\triangleright^\otimes$ follows from the standardization theorem.

The three notions of evaluation are related as follows:

Fact VII.1.15

In λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow , we have

$$\boxed{t \triangleright^\otimes} \iff \boxed{\begin{array}{c} t \overset{h}{\triangleright}^\otimes \\ t \overset{a}{\triangleright}^\otimes \end{array}} \iff \boxed{\begin{array}{c} t \overset{lo}{\triangleright}^\otimes \\ t \neg\triangleright^\otimes \end{array}}$$

where propositions in the same nodes are equivalent, and both implications are strict.

Proof

- $\boxed{t \overset{h}{\triangleright}^\otimes} \Leftrightarrow \boxed{t \overset{a}{\triangleright}^\otimes}$ and $\boxed{t \overset{lo}{\triangleright}^\otimes} \Leftrightarrow \boxed{t \neg\triangleright^\otimes}$ By the previous fact.

VII. Call-by-name solvability

- $\boxed{t \triangleright^{\otimes} \stackrel{\text{strict}}{\Leftarrow} t \stackrel{\text{h}\triangleright^{\otimes}}{\Leftarrow} t \stackrel{\text{strict}}{\Leftarrow} t \stackrel{\text{lo}\triangleright^{\otimes}}{\Leftarrow}}$ Since all three reductions are deterministic, the contrapositive is equivalent to $t \triangleright^{\omega} \Rightarrow t \stackrel{\text{h}\triangleright^{\omega}}{\Rightarrow} t \stackrel{\text{lo}\triangleright^{\omega}}{\Rightarrow}$, which immediately follows from $\triangleright \subseteq \stackrel{\text{h}}{\triangleright} \subseteq \stackrel{\text{lo}}{\triangleright}$.
- $\boxed{t \triangleright^{\otimes} \not\Rightarrow t \stackrel{\text{h}\triangleright^{\otimes}}{\triangleright}}$ The expression $\lambda x^N. \Omega_N$ (resp. command $\langle \mu(x^n \cdot \star^n). \Omega_n | \star^n \rangle$) it a counter-example in λ_N^{\rightarrow} (resp. $\text{Li}_n^{\rightarrow}$).
- $\boxed{t \stackrel{\text{h}\triangleright^{\otimes}}{\triangleright} \not\Rightarrow t \stackrel{\text{lo}\triangleright^{\otimes}}{\triangleright}}$ The expression $x^N \Omega_N$ (resp. command $\langle x^n | \Omega_n \cdot \star^n \rangle$) it a counter-example in λ_N^{\rightarrow} (resp. $\text{Li}_n^{\rightarrow}$).

Existence of normal forms and evaluation Evaluation under \triangleright^{\otimes} (resp. $\stackrel{\text{h}}{\triangleright^{\otimes}}$) can be studied indirectly via the existence of an \triangleright -normal (resp. $\stackrel{\text{h}}{\triangleright}$ -normal) form:

Fact VII.1.16

In λ_N^{\rightarrow} , λ_n^{\rightarrow} , $\text{Li}_n^{\rightarrow}$, and L_n^{\rightarrow} , for each reduction $\rightsquigarrow \in \{\triangleright, \stackrel{\text{h}}{\triangleright}, \stackrel{\text{a}}{\triangleright}, \stackrel{\text{lo}}{\triangleright}, \rightarrow\}$, we have

$$\text{dom}(\rightsquigarrow^{\otimes}) = \text{dom}(\rightarrow^* \rightsquigarrow) = \text{dom}(\approx_{\beta} \rightsquigarrow)$$

i.e. for any object t , we have

$$(\exists O', t \rightsquigarrow^* t' \rightsquigarrow) \Leftrightarrow (\exists O', t \rightarrow^* t' \rightsquigarrow) \Leftrightarrow (\exists O', t \approx_{\beta} t' \rightsquigarrow)$$

In particular, in λ_N^{\rightarrow} , we have

$$(\exists T'_N \in \mathbf{T}_N^{\text{ONF}}, T_N \triangleright^* T'_N) \Leftrightarrow (\exists T'_N \in \mathbf{T}_N^{\text{ONF}}, T_N \rightarrow^* T'_N) \Leftrightarrow (\exists T'_N \in \mathbf{T}_N^{\text{ONF}}, T_N \approx_{\beta} T'_N)$$

and

$$(\exists T'_N \in \mathbf{T}_N^{\text{AHNF}}, T_N \triangleright^* T'_N) \Leftrightarrow (\exists T'_N \in \mathbf{T}_N^{\text{AHNF}}, T_N \rightarrow^* T'_N) \Leftrightarrow (\exists T'_N \in \mathbf{T}_N^{\text{AHNF}}, T_N \approx_{\beta} T'_N)$$

Proof

- $\boxed{\text{dom}(\rightsquigarrow^{\otimes}) = \text{dom}(\rightarrow^* \rightsquigarrow)}$ The \subseteq inclusion is trivial by $\rightsquigarrow \subseteq \rightarrow$. Since $\stackrel{\text{h}}{\triangleright^{\otimes}} = \stackrel{\text{a}}{\triangleright^{\otimes}}$ and $\stackrel{\text{lo}}{\triangleright^{\otimes}} = \rightarrow^{\otimes}$, it suffices to show the \supseteq inclusion for $\rightsquigarrow \in \{\triangleright, \stackrel{\text{h}}{\triangleright}, \rightarrow\}$. For $\rightsquigarrow = \rightarrow$, this inclusion is trivial. Now, suppose that $t \rightarrow^* t'' \rightsquigarrow$. By Δ (resp. Δ), we have $t \triangleright^* t' \rightarrow^* t''$ (resp. $t \stackrel{\text{h}}{\triangleright^*} t' \rightarrow^* t''$) for some t' . Since \triangleright -reducibility (resp. $\stackrel{\text{h}}{\triangleright}$ -reducibility) is preserved by \rightarrow (resp. $\stackrel{\text{h}}{\triangleright}$), t' is necessarily \triangleright -normal (resp. $\stackrel{\text{h}}{\triangleright}$ -normal), and we can therefore conclude that $t \triangleright^{\otimes} t'$ (resp. $t \stackrel{\text{h}}{\triangleright^{\otimes}} t'$).
- $\boxed{\text{dom}(\rightarrow^* \rightsquigarrow) = \text{dom}(\approx_{\beta} \rightsquigarrow)}$ The \subseteq inclusion is trivial by $\rightarrow^* \subseteq \approx_{\beta}$. Now suppose that $t_{\odot} \approx_{\beta} t_{\bullet} \rightsquigarrow$. By confluence of \rightarrow , we have $t_{\odot} \rightarrow^* t' \leftarrow^* t_{\bullet}$ for some t' . Since \rightsquigarrow -normal forms are preserved by \rightarrow^a , t' is \rightsquigarrow -normal, and we are done.

^aFor $\rightsquigarrow = \rightarrow$, this holds vacuously: given an \rightsquigarrow -normal normal expression, all expressions it \rightarrow -

VII. *Call-by-name solvability*

reduces to (of which there are none) are \rightsquigarrow -normal.

Figure VII.1.3: Normal forms in λ_N^\rightarrow , $\underline{\lambda}_n^\rightarrow$, Li_n^\rightarrow , and L_n^\rightarrow

 Figure VII.1.3.a: Normal forms in λ_N^\rightarrow

 Operational normal forms in λ_N^\rightarrow :

$$\mathbf{T}_N^{\text{ONF}} \ni \quad T_N^{\text{ONF}} ::= \lambda x^N. T_N \mid x^N T_N^1 \dots T_N^r$$

 Ahead normal forms in λ_N^\rightarrow :

$$\mathbf{T}_N^{\text{AHNF}} \ni \quad T_N^{\text{AHNF}} ::= \lambda x^N. T_N^{\text{AHNF}} \mid x^N T_N^1 \dots T_N^r$$

 (Strong) normal forms in λ_N^\rightarrow :

$$\mathbf{T}_N^{\text{NF}} \ni \quad T_N^{\text{NF}} ::= \lambda x^N. T_N^{\text{NF}} \mid x^N T_N^{\text{NF},1} \dots T_N^{\text{NF},r}$$

 Figure VII.1.3.b: Normal forms in (inside-out) $\underline{\lambda}_n^\rightarrow$

 Operational normal forms in $\underline{\lambda}_n^\rightarrow$:

$$\begin{aligned} \mathbf{c}_n^{\text{ONF}} &\ni \quad c_n^{\text{ONF}} ::= \lambda x^n. c_n \mid \mathbb{S}_n \boxed{x^n} \\ \mathbf{t}_n^{\text{ONF}} &\ni \quad t_n^{\text{ONF}} ::= t_n \\ \mathbf{s}_n^{\text{ONF}} &\ni \quad s_n^{\text{ONF}} ::= \mathbb{S}_n \\ \mathbf{e}_n^{\text{ONF}} &\ni \quad e_n^{\text{ONF}} ::= e_n \end{aligned}$$

 Ahead normal forms in $\underline{\lambda}_n^\rightarrow$:

$$\begin{aligned} \mathbf{c}_n^{\text{AHNF}} &\ni \quad c_n^{\text{AHNF}} ::= \lambda x^n. c_n^{\text{AHNF}} \mid \mathbb{S}_n \boxed{x^n} \\ \mathbf{t}_n^{\text{AHNF}} &\ni \quad t_n^{\text{AHNF}} ::= x^n \mid \lambda x^n. c_n^{\text{AHNF}} \mid \text{ctot}(c_n^{\text{AHNF}}) \\ \mathbf{s}_n^{\text{AHNF}} &\ni \quad s_n^{\text{AHNF}} ::= \mathbb{S}_n \\ \mathbf{e}_n^{\text{AHNF}} &\ni \quad e_n^{\text{AHNF}} ::= e_n \end{aligned}$$

 (Strong) normal forms in $\underline{\lambda}_n^\rightarrow$:

$$\begin{aligned} \mathbf{c}_n^{\text{NF}} &\ni \quad c_n^{\text{NF}} ::= \lambda x^n. c_n^{\text{NF}} \mid \mathbb{S}_n^{\text{NF}} \boxed{x^n} \\ \mathbf{t}_n^{\text{NF}} &\ni \quad t_n^{\text{NF}} ::= x^n \mid \lambda x^n. c_n^{\text{NF}} \mid \text{ctot}(c_n^{\text{NF}}) \\ \mathbf{s}_n^{\text{NF}} &\ni \quad s_n^{\text{NF}} ::= \square \mid \mathbb{S}_n^{\text{NF}} \boxed{t_n^{\text{NF}}} \\ \mathbf{e}_n^{\text{NF}} &\ni \quad e_n^{\text{NF}} ::= \mathbb{S}_n^{\text{NF}} \mid \text{let } x^n := \square \text{ in } c_n^{\text{NF}} \end{aligned}$$

 Figure VII.1.3.c: Normal forms in Li_n^\rightarrow

 Same grammar as for L_n^\rightarrow , with every stack variable α^n replaced by \star^n

Figure VII.1.3.d: Normal forms in L_n^\rightarrow

Operational normal forms in L_n^\rightarrow :

$$\begin{array}{ll} \mathbf{c}_n^{\text{ONF}} \ni & c_n^{\text{ONF}} ::= \langle \mu(x^n \cdot \alpha^n).c \mid \beta^n \rangle \mid \langle x^n \mid s_n \rangle \\ \mathbf{t}_n^{\text{ONF}} \ni & t_n^{\text{ONF}} ::= t_n \\ \mathbf{s}_n^{\text{ONF}} \ni & s_n^{\text{ONF}} ::= s_n \\ \mathbf{e}_n^{\text{ONF}} \ni & e_n^{\text{ONF}} ::= e_n \end{array}$$

Ahead normal forms in L_n^\rightarrow :

$$\begin{array}{ll} \mathbf{c}_n^{\text{AHNF}} \ni & c_n^{\text{AHNF}} ::= \langle \mu(x^n \cdot \alpha^n).c_n^{\text{AHNF}} \mid \alpha^n \rangle \mid \langle x^n \mid s_n \rangle \\ \mathbf{t}_n^{\text{AHNF}} \ni & t_n^{\text{AHNF}} ::= \mu(x^n \cdot \alpha^n).c_n^{\text{AHNF}} \mid \mu\alpha^n.c_n^{\text{AHNF}} \mid x^n \\ \mathbf{s}_n^{\text{AHNF}} \ni & s_n^{\text{AHNF}} ::= s_n \\ \mathbf{e}_n^{\text{AHNF}} \ni & e_n^{\text{AHNF}} ::= e_n \end{array}$$

(Strong) normal forms in L_n^\rightarrow :

$$\begin{array}{ll} \mathbf{c}_n^{\text{NF}} \ni & c_n^{\text{NF}} ::= \langle \mu(x^n \cdot \alpha^n).c_n^{\text{NF}} \mid \beta^n \rangle \mid \langle x^n \mid s_n^{\text{NF}} \rangle \\ \mathbf{t}_n^{\text{NF}} \ni & t_n^{\text{NF}} ::= x^n \mid \mu\alpha^n.c_n^{\text{NF}} \mid \mu(x^n \cdot \alpha^n).c_n^{\text{NF}} \\ \mathbf{s}_n^{\text{NF}} \ni & s_n^{\text{NF}} ::= \alpha^n \mid t_n^{\text{NF}} \cdot s_n^{\text{NF}} \\ \mathbf{e}_n^{\text{NF}} \ni & e_n^{\text{NF}} ::= s_n^{\text{NF}} \mid \tilde{\mu}x^n.c_n^{\text{NF}} \end{array}$$

VII.2. Usefulness of the trivial interpretation of programs

VII.3. Instanciating the general definitions

VII.3.1. Parameters

In λ_N^\rightarrow , (resp. λ_n^\rightarrow , Li_n^\rightarrow , L_n^\rightarrow), there are many reasonable ways of instanciating the general definitions by choosing a reduction \rightsquigarrow , and sets of programs \mathcal{P} , of contexts \mathcal{K} , of inputs \mathcal{J} , and of outputs \mathcal{O} . For the values of these parameters we consider, many notions end up being equivalent, and the only relevant parameter is the reduction \rightsquigarrow . For each definition, we therefore give our preferred definition that only depends on \rightsquigarrow , then generalize by making it depend on the other parameters, and finally show that for many values of the parameters, the instances of the general definitions are equivalent to our preferred definition.

Terms and states We define terms (i.e. program fragments without holes) and states (i.e. things we want to reduce) in λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow as follows:

Definition VII.3.1: Terms and states

Terms are all objects of the syntax:

$$\mathcal{T}_{\text{term}} = \mathbf{T}_N \text{ in } \lambda_N^\rightarrow, \quad \text{and} \quad \mathcal{T}_{\text{term}} = \mathbf{c}_n \cup \mathbf{t}_n \cup \mathbf{e}_n \text{ in } \lambda_n^\rightarrow, Li_n^\rightarrow, \text{ and } L_n^\rightarrow$$

States are expressions in λ_N^\rightarrow , and commands in the other calculi:

$$\mathcal{S}_{\text{state}} = \mathbf{T}_N \text{ in } \lambda_N^\rightarrow, \quad \text{and} \quad \mathcal{S}_{\text{state}} = \mathbf{c}_n \text{ in } \lambda_n^\rightarrow, Li_n^\rightarrow, \text{ and } L_n^\rightarrow$$

These sets of $\mathcal{T}_{\text{term}}$ and $\mathcal{S}_{\text{state}}$ will be used to define, claim and prove things in all four calculi λ_N^\rightarrow , λ_n^\rightarrow , Li_n^\rightarrow , and L_n^\rightarrow at once so as to avoid unnecessary duplications.

Closedness In λ_N^\rightarrow , a natural choice for the set of programs \mathcal{P} is the set of closed expressions. In L-calculi, it makes no sense to consider closed commands, e.g. because there are no closed commands in Li_n^\rightarrow and L_n^\rightarrow (because any command has a free stack variable). This could be solved by extending the calculi with a stack constant \wedge (with no associated reduction rules) and substituting all free stack variables by \wedge . Doing so explicitly is mostly useless since for each reduction \rightsquigarrow we consider, we would have

$$c_n \rightsquigarrow^\oplus c'_n \Leftrightarrow c_n[\wedge/\alpha_1^n, \dots, \wedge/\alpha_q^n] \rightsquigarrow^\oplus c'_n[\wedge/\alpha_1^n, \dots, \wedge/\alpha_q^n]$$

and we therefore leave this substitution implicit and call virtually closed the commands that would be closed after such a substitution:

Definition VII.3.2

A state is said to be *virtually closed* when it has no free value variables. The set of virtually closed states is denoted by

$$\dot{\mathcal{S}}_{\text{state}} \stackrel{\text{def}}{=} \{q \in \mathcal{S}_{\text{state}} \mid \text{FV}_v(q) = \emptyset\}$$

VII. Call-by-name solvability

Parameters While these are not the only reasonable choices, we restrict our attention the the following values for the parameters:

- The set of programs \mathcal{P} can be:
 - the set of all states, i.e. of expressions in λ_N^{\rightarrow} ; and of commands in the other calculi:

$$\mathcal{P} = \mathcal{S}_{\text{state}} = \mathbf{T}_N \text{ in } \lambda_N^{\rightarrow}, \quad \text{and} \quad \mathcal{P} = \mathcal{S}_{\text{state}} = \mathbf{c}_n \text{ in } \lambda_n^{\rightarrow}, \text{ Li}_n^{\rightarrow}, \text{ and } L_n^{\rightarrow}$$

- the set of virtually closed states, i.e. of closed expressions in λ_N^{\rightarrow} ; and of commands with no free value variables x^n in the other calculi:

$$\mathcal{P} = \mathcal{S}_{\text{state}}^{\dots} = \overline{\mathbf{T}_V} \text{ in } \lambda_V^{\rightarrow}, \quad \text{and} \quad \mathcal{P} = \mathcal{S}_{\text{state}}^{\dots} = \overline{\mathbf{c}_n} \text{ in } \lambda_n^{\rightarrow}, \text{ Li}_n^{\rightarrow}, \text{ and } L_n^{\rightarrow}$$

- The set of inputs \mathcal{I}_{put} can be:
 - the trivial set of inputs $\mathcal{I} = \mathcal{I}_{\text{put},\bullet}$, with

$$\mathcal{I}_{\text{put},\bullet} = \{\bullet\}, \quad \text{and} \quad \text{initial_state}_{\bullet}(\textcolor{blue}{p}, \bullet) = \textcolor{blue}{p}$$

- the set of stacks $\mathcal{I} = \mathcal{I}_{\text{put}_S}$, with

$$\mathcal{I}_{\text{put}_S} = \mathbf{S}_N \quad \text{and} \quad \text{initial_state}_S(\textcolor{blue}{T}_N, \mathbf{S}_N) = \mathbf{S}_N \boxed{\textcolor{blue}{T}_N} \quad \text{in } \lambda_N^{\rightarrow}$$

$$\mathcal{I}_{\text{put}_S} = \mathbf{s}_n \quad \text{and} \quad \text{initial_state}_S(\textcolor{blue}{t}_n, \mathbf{s}_n) = \mathbf{s}_n \boxed{\textcolor{blue}{t}_n} \quad \text{in } \lambda_n^{\rightarrow}$$

$$\mathcal{I}_{\text{put}_S} = \mathbf{s}_n \quad \text{and} \quad \text{initial_state}_S(\textcolor{blue}{t}_n, \textcolor{blue}{s}_n) = \langle \textcolor{blue}{t}_n | \textcolor{blue}{s}_n \rangle \quad \text{in } \text{Li}_n^{\rightarrow} \text{ and } L_n^{\rightarrow}$$

- the set of virtually closed stacks:

$$\mathcal{I}_{\text{put}_{S-X}} = \overline{\mathbf{S}_N} \text{ in } \lambda_N^{\rightarrow} \quad \text{and} \quad \mathcal{I}_{\text{put}_{S-X}} = \overline{\mathbf{s}_n} \text{ in } \lambda_n^{\rightarrow}, \text{ Li}_n^{\rightarrow} \text{ and } L_n^{\rightarrow}$$

with

$$\text{initial_state}_{S-X} \stackrel{\text{def}}{=} \text{initial_state}_S$$

- The set of outputs \mathcal{O} can be:

- trivial:

$$\mathcal{O} = \mathcal{O}_{\text{output},\bullet} = \{\bullet\}, \quad \text{and} \quad \text{output}_{\bullet}(\textcolor{blue}{q}) = \bullet$$

- terms quotiented by an equivalence relation \mathcal{R} :

$$\mathcal{O} = \mathcal{O}_{\text{output}_{\mathcal{R}}} = \mathcal{T}_{\text{erm}} / \mathcal{R}, \quad \text{and} \quad \text{output}_{\mathcal{R}}(\textcolor{blue}{q}) = \{t \in \mathcal{T}_{\text{erm}} \mid \textcolor{blue}{q} \mathcal{R} t\}$$

- The set of contexts \mathcal{K} can be:

- the set of all contexts:

$$\mathcal{K} = \mathbf{K}_N \text{ in } \lambda_N^{\rightarrow}, \quad \text{and} \quad \mathcal{K} = \mathbf{k}_n \text{ in } \lambda_n^{\rightarrow}, \text{ Li}_n^{\rightarrow} \text{ and } L_n^{\rightarrow}$$

- the set of all function contexts in λ_V^{\rightarrow} ⁹:

$$\mathcal{K} = \mathbf{F}_N$$

where

$$\mathbf{F}_N \stackrel{\text{def}}{=} \{(\lambda \textcolor{blue}{x}_1^N \dots \lambda \textcolor{blue}{x}_q^N \cdot \square) T_N^1 \dots T_N^r \mid (\textcolor{blue}{x}_1^N, \dots, \textcolor{blue}{x}_q^N) \in \mathcal{V}^q \text{ and } (T_N^1, \dots, T_N^r) \in \mathbf{T}_N^r\}$$

⁹These could also be defined in the other calculi, but there is no point in doing so since disubstitutions are more natural.

VII. Call-by-name solvability

- the set of all applied function contexts in λ_V^- :

$$\mathcal{K} = \mathbf{F}_V^a$$

where

$$\mathbf{F}_N^a = \{(\lambda x_1^N \dots \lambda x_q^N \cdot \square) T_N^1 \dots T_N^q \mid (x_1^N, \dots, x_q^N) \in \mathcal{V}^q \text{ and } (T_N^1, \dots, T_N^q) \in \mathbf{T}_N^r \text{ with } q \leq r\}$$

- the set of disubstitutions

$$\begin{aligned} \mathcal{K} &= \varphi_N = \{(\sigma, \square T_N^1 \dots T_N^q) \mid \sigma : \mathcal{V} \xrightarrow{\text{fin}} \mathbf{T}_N \text{ and } (T_N^1, \dots, T_N^q) \in \mathbf{T}_N^q\} && \text{in } \lambda_N^- \\ \mathcal{K} &= \varphi_n = \{(\sigma, \square t_n^1 \dots t_n^q) \mid \sigma : \mathcal{V} \xrightarrow{\text{fin}} \mathbf{t}_n \text{ and } (t_n^1, \dots, t_n^q) \in \mathbf{t}_n^q\} && \text{in } \lambda_n^- \\ \mathcal{K} &= \varphi_n = \{\sigma \cup (\star^v \mapsto t_n^1 \cdot \dots \cdot t_n^q \cdot \star^v) \mid \sigma : \mathcal{V} \xrightarrow{\text{fin}} \mathbf{t}_n \text{ and } (t_n^1, \dots, t_n^q) \in \mathbf{t}_n^q\} && \text{in } \text{Li}_n^- \\ \mathcal{K} &= \varphi_n = \{\sigma \cup \psi \mid \sigma : \mathcal{V} \xrightarrow{\text{fin}} \mathbf{t}_n \text{ and } \psi : \mathcal{S} \xrightarrow{\text{fin}} \mathbf{s}_v\} && \text{in } L_n^- \end{aligned}$$

The main definitions correspond to the choices

$$\mathcal{P} = \mathcal{S}_{\text{state}}, \mathcal{K} = \varphi, \text{ and } \mathcal{F} = \mathcal{O} = \{\bullet\}$$

VII.3.2. A meaning for programs

Main definition The main definition only tests for convergence with respect to a reduction \rightsquigarrow :

Definition VII.3.3: Halting preorder and equivalence

Given a reduction \rightsquigarrow , the *halting preorder* $\lesssim_{\rightsquigarrow}$ is defined on states (i.e. expressions in λ_N^- , and commands in the other calculi) by:

$$q_1 \lesssim_{\rightsquigarrow} q_2 \stackrel{\text{def}}{=} q_1 \rightsquigarrow^{\otimes} \Rightarrow q_2 \rightsquigarrow^{\otimes}$$

The *halting equivalence* \sim_{\rightsquigarrow} , and the *strict halting preorder* $<_{\rightsquigarrow}$ are respectively the equivalence relation and the strict preorder induced by the halting preorder $\lesssim_{\rightsquigarrow}$:

$$\begin{aligned} q_1 \sim_{\rightsquigarrow} q_2 &\stackrel{\text{def}}{=} q_1 \lesssim_{\rightsquigarrow} q_2 \text{ and } q_1 \gtrsim_{\rightsquigarrow} q_2 && (\text{i.e. } q_1 \rightsquigarrow^{\otimes} \Leftrightarrow q_2 \rightsquigarrow^{\otimes}) \\ q_1 <_{\rightsquigarrow} q_2 &\stackrel{\text{def}}{=} q_1 \lesssim_{\rightsquigarrow} q_2 \text{ and } q_1 \not\gtrsim_{\rightsquigarrow} q_2 && (\text{i.e. } q_1 \rightsquigarrow^{\otimes} \text{ and } q_2 \not\rightsquigarrow^{\otimes}) \end{aligned}$$

Alternative definitions Alternative definitions allow for other sets of inputs \mathcal{F} or \mathcal{O} :

Definition VII.3.4

Given a set of inputs \mathcal{F} , a reduction \rightsquigarrow , and a set of outputs \mathcal{O} , we define

$$\begin{aligned} \llbracket p \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} : \mathcal{F} &\rightarrow \mathcal{O} \\ i &\mapsto \begin{cases} \text{output}(q) & \text{if } \text{initial_state}(p, i) \rightsquigarrow^* q \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

VII. Call-by-name solvability

We write $p_1 \lesssim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} p_2$ for $\llbracket p_1 \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} \subseteq \llbracket p_2 \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$. The induced equivalence (resp. strict preorder) is denoted by $\sim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$ (resp. $<_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$):

$$\begin{aligned} q_1 \sim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} q_2 &\stackrel{\text{def}}{=} q_1 \lesssim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} q_2 \text{ and } q_1 \gtrsim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} q_2 & (\text{i.e. } \llbracket p_1 \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} = \llbracket p_2 \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}) \\ q_1 <_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} q_2 &\stackrel{\text{def}}{=} q_1 \lesssim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} q_2 \text{ and } q_1 \not\gtrsim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} q_2 & (\text{i.e. } \llbracket p_1 \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}} \subsetneq \llbracket p_2 \rrbracket_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}) \end{aligned}$$

To keep the subscripts short, we sometimes simplify $\mathcal{I}_{\text{input}}$ and $\mathcal{O}_{\text{output}}$ to \bullet , $\mathcal{I}_{\text{input}_S}$ to S , $\mathcal{I}_{\text{input}_S \neg x}$ to $S \neg x$, and $\mathcal{O}_{\text{output}_R}$ to R , so that e.g.

$$\lesssim_{\bullet, \rightsquigarrow, \bullet} \stackrel{\text{ntn}}{=} \lesssim_{\mathcal{I}_{\text{input}}, \rightsquigarrow, \mathcal{O}_{\text{output}}}$$

Fact VII.3.5

We have $\lesssim_{\rightsquigarrow} = \lesssim_{\bullet, \rightsquigarrow, \bullet}$.

Proof

Immediate.

Properties The halting preorder $\lesssim_{\rightsquigarrow}$ (resp. halting equivalence \sim_{\rightsquigarrow}) is of course a preorder (resp. equivalence):

Fact VII.3.6

For any \mathcal{F} , \rightsquigarrow , and \mathcal{O} , the relation $\lesssim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$ is a preorder, $\sim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$ is an equivalence relation, and $<_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$ is a strict preorder. In particular, $\lesssim_{\rightsquigarrow}$ is a preorder, \sim_{\rightsquigarrow} an equivalence relation, and $<_{\rightsquigarrow}$ a strict preorder.

Proof

Immediate.

Note that $\lesssim_{\mathcal{F}, \rightsquigarrow, \mathcal{O}}$ only depends on \rightsquigarrow through $\rightsquigarrow^{\otimes}$, so that any reductions inducing the same notion of evaluation $\rightsquigarrow^{\otimes}$ also induce the same halting preorder:

Fact VII.3.7

For any \mathcal{F} , \rightsquigarrow , and \mathcal{O} , we have

$$\rightsquigarrow_1^{\otimes} = \rightsquigarrow_2^{\otimes} \Rightarrow \lesssim_{\mathcal{F}, \rightsquigarrow_1, \mathcal{O}} = \lesssim_{\mathcal{F}, \rightsquigarrow_2, \mathcal{O}} \Rightarrow \sim_{\mathcal{F}, \rightsquigarrow_1, \mathcal{O}} = \sim_{\mathcal{F}, \rightsquigarrow_2, \mathcal{O}}$$

In particular,

$$\lesssim_{\text{hp}} = \lesssim_{\text{hp}} \quad \text{and} \quad \lesssim_{\text{hp}} = \lesssim_{\rightarrow}$$

VII. Call-by-name solvability

Proof

Both \Rightarrow implications are by definition. The two particular equalities follow from the equalities $\xrightarrow{h}_{\triangleright}^{\otimes} = \xrightarrow{a}_{\triangleright}^{\otimes}$ and $\xrightarrow{lo}_{\triangleright}^{\otimes} = \xrightarrow{-}_{\triangleright}^{\otimes}$ (Fact VII.1.14).

For the reductions \rightsquigarrow we consider, the halting equivalence \sim_{\rightsquigarrow} contains the β -equivalence \approx_{β} :

Fact VII.3.8

We have:

$$\approx_{\beta} \subseteq \sim_{\triangleright}, \quad \approx_{\beta} \subseteq \sim_{\vdash}, \quad \text{and} \quad \approx_{\beta} \subseteq \sim_{\rightarrow}$$

Proof

This is an immediate consequence of the equivalence between \rightsquigarrow^* -reducing to an \rightsquigarrow -normal form and having an \rightsquigarrow -normal form modulo \approx_{β} (Fact VII.1.16). Indeed, if $t_1 \approx_{\beta} t_2$ then

$$t_1 \rightsquigarrow^{\otimes} \Leftrightarrow t_1 \approx_{\beta} \rightsquigarrow \Leftrightarrow t_2 \approx_{\beta} \rightsquigarrow \Leftrightarrow t_2 \rightsquigarrow^{\otimes}$$



VII.3.3. Observational preorder and equivalence

Main definitions

The main notions of observational preorder and equivalence are defined on states using disubstitutions, and then extended to non-state expressions by embedding them into a states:

Definition VII.3.9: Observational preorder and equivalence

Given a reduction \rightsquigarrow , the \rightsquigarrow -observational preorder $\sqsubseteq_{\rightsquigarrow}$ is defined on states by

$$q_1 \sqsubseteq_{\rightsquigarrow} q_2 \stackrel{\text{def}}{=} \forall \varphi, q_1[\varphi] \lesssim_{\rightsquigarrow} q_2[\varphi]$$

In λ_N^{\rightarrow} , a disubstitution $\varphi = (\sigma, \mathbb{S}_N)$ is just a substitution σ and a stack \mathbb{S}_N so that this simplifies to

$$T_N^1 \sqsubseteq_{\rightsquigarrow} T_N^2 \stackrel{\text{def}}{=} \forall \sigma, \forall \mathbb{S}_N, \mathbb{S}_N[T_N^1[\sigma]] \lesssim_{\rightsquigarrow} \mathbb{S}_N[T_N^2[\sigma]]$$

VII. Call-by-name solvability

In λ_n^\rightarrow , Li_n^\rightarrow and L_n^\rightarrow , the operational preorder \sqsubseteq_∞ is extended to expressions and evaluation contexts as follows:

- In λ_n^\rightarrow :

$$\begin{aligned} t_n^1 \sqsubseteq_\infty t_n^2 &\stackrel{\text{def}}{=} \underline{t_n^1} \sqsubseteq_\infty \underline{t_n^2} \\ e_n^1 \sqsubseteq_\infty e_n^2 &\stackrel{\text{def}}{=} \forall x^n, \underline{e_n^1[x^n]} \sqsubseteq_\infty \underline{e_n^2[x^n]} \end{aligned}$$

- In Li_n^\rightarrow :

$$\begin{aligned} t_n^1 \sqsubseteq_\infty t_n^2 &\stackrel{\text{def}}{=} \langle t_n^1 | \star^n \rangle \sqsubseteq_\infty \langle t_n^2 | \star^n \rangle \\ e_n^1 \sqsubseteq_\infty e_n^2 &\stackrel{\text{def}}{=} \forall x^n, \langle x^n | e_n^1 \rangle \sqsubseteq_\infty \langle x^n | e_n^2 \rangle \end{aligned}$$

- In L_n^\rightarrow :

$$\begin{aligned} t_n^1 \sqsubseteq_\infty t_n^2 &\stackrel{\text{def}}{=} \forall \alpha^n, \langle t_n^1 | \alpha^n \rangle \sqsubseteq_\infty \langle t_n^2 | \alpha^n \rangle \\ e_n^1 \sqsubseteq_\infty e_n^2 &\stackrel{\text{def}}{=} \forall x^n, \langle x^n | e_n^1 \rangle \sqsubseteq_\infty \langle x^n | e_n^2 \rangle \end{aligned}$$

The \rightsquigarrow -observational equivalence \approx_∞ , and the strict \rightsquigarrow -observational preorder \sqsubseteq_∞ are respectively the equivalence relation and the strict preorder induced by the observational preorder \sqsubseteq_∞ :

$$\begin{aligned} t_1 \approx_\infty t_2 &\stackrel{\text{def}}{=} t_1 \sqsubseteq_\infty t_2 \text{ and } t_1 \sqsupseteq_\infty t_2 \text{ (i.e. } \forall \varphi, q_1[\varphi] \sim_\infty q_2[\varphi]) \\ t_1 \sqsubset_\infty t_2 &\stackrel{\text{def}}{=} t_1 \sqsubseteq_\infty t_2 \text{ and } t_1 \not\sqsupseteq_\infty t_2 \end{aligned}$$

We sometimes add a “c” (resp. “i”) subscript to emphasize that the ambient calculus is L_n^\rightarrow (resp. is λ_n^\rightarrow , λ_n^\rightarrow , or Li_n^\rightarrow), e.g. writing $\sqsubseteq_{c,\infty}$ for the observational preorder of L_n^\rightarrow (resp. $\sqsubseteq_{i,\infty}$ for the observational preorder of λ_n^\rightarrow , λ_n^\rightarrow , or Li_n^\rightarrow).

Since it only tests for convergence via \lesssim_∞ , the observational equivalence \approx_∞ could a priori be too loose but it is not; it is able to distinguish between many expressions that we could have wanted to use as outputs:

Example VII.3.10

In λ_N^\rightarrow , for any reduction \rightsquigarrow such that $\triangleright \subseteq \rightsquigarrow \subseteq \triangleright$, we have:

- $\boxed{x^N \not\approx_\infty y^N}$ Two distinct variables x^N and y^N are never \rightsquigarrow -observationally equivalent. Indeed, we have

$$x^N[I_N/x^N, \Omega_N/y^N] = I_N >_\infty \Omega_N = y^N[I_N/x^N, \Omega_N/y^N]$$

- $\boxed{\lambda x^N. \lambda y^N. x^N \not\approx_\infty \lambda x^N. \lambda y^N. y^N}$ The Church encoding of booleans $\lambda x^N. \lambda y^N. x^N$ and $\lambda x^N. \lambda y^N. y^N$ are not \rightsquigarrow -observationally equivalent. Indeed, for $\mathbb{S}_N = \square I_N \Omega_N$, we have

$$\mathbb{S}_N \boxed{\lambda x^N. \lambda y^N. x^N} \rightsquigarrow^* I_N >_\infty \Omega_N \leftarrow^* \mathbb{S}_N \boxed{\lambda x^N. \lambda y^N. y^N}$$

- $\boxed{\ulcorner n \urcorner \not\approx_\infty \ulcorner m \urcorner}$ The Church encodings of two distinct natural numbers are not \rightsquigarrow -observationally equivalent. Indeed, WLOG $n < m$ and for

$$\mathbb{S}_N = \square K_N I_N T_N^1 \dots T_N^n \Omega_N I_N \dots I_N$$

VII. Call-by-name solvability

with T_N^1, \dots, T_N^n arbitrary, and $m - n$ copies of I_N after Ω_N , we get

$$\mathbb{S}_N \llbracket m \rrbracket \rightsquigarrow^* I_N \succsim \Omega_N I_N \dots I_N \leftarrow^* \mathbb{S}_N \llbracket n \rrbracket$$

In λ_n^\rightarrow , Li_n^\rightarrow and L_n^\rightarrow , the extension to expressions and evaluation contexts could also be define by using a quantification on values (i.e. expressions) and stacks in place of value and stack variables:

Fact VII.3.11

We have:

• In λ_n^\rightarrow :

$$t_n^1 \sqsubseteq_{\rightsquigarrow} t_n^2 \Leftrightarrow \forall \mathbb{S}_n, \mathbb{S}_n \llbracket t_n^1 \rrbracket \sqsubseteq_{\rightsquigarrow} \mathbb{S}_n \llbracket t_n^2 \rrbracket$$

$$e_n^1 \sqsubseteq_{\rightsquigarrow} e_n^2 \Leftrightarrow \forall v_n, e_n^1 \llbracket v_n \rrbracket \sqsubseteq_{\rightsquigarrow} e_n^2 \llbracket v_n \rrbracket$$

• In Li_n^\rightarrow and L_n^\rightarrow :

$$t_n^1 \sqsubseteq_{\rightsquigarrow} t_n^2 \Leftrightarrow \forall \mathbb{S}_n, \langle t_n^1 | \mathbb{S}_n \rangle \sqsubseteq_{\rightsquigarrow} \langle t_n^2 | \mathbb{S}_n \rangle$$

$$e_n^1 \sqsubseteq_{\rightsquigarrow} e_n^2 \Leftrightarrow \forall v_n, \langle v_n | e_n^1 \rangle \sqsubseteq_{\rightsquigarrow} \langle v_n | e_n^2 \rangle$$

(where the quantifications $\forall v_n$ could also be written $\forall t_n$ since values are exactly expressions in call-by-name).

Proof

The \Leftarrow implication follows from value (resp. stack) variables being values (resp. stacks). The \Rightarrow implication follows from the possibility of choosing a fresh value (resp. stack) variable, and of freely extending the disubstitution to it.

We prefer the definition with variables because it does not need to be adapted if we restrict the quantification $\forall \varphi$ in Definition VII.3.9 to some subset of disubstitutions (e.g. in call-by-value where we sometimes want to restrict the quantification to simple disubstitutions).

Alternative definitions

In order to keep the definitions of the observational preorder short, we defined it directly on terms t , but since we only want it to relate expressions to expressions, stacks to stacks, etc., we define the notion of compatible terms:

Definition VII.3.12

Two terms t_1 and t_2 are said to be *compatible*, written $t_1 \equiv t_2$, when they are both expressions, both evaluation contexts, or both commands.

We now give alternative definitions of the observational preorder that use the more general $\lesssim_{\mathcal{J}, \dots, \mathcal{G}}$ in place of $\lesssim_{\rightsquigarrow}$, and also allow varying the set of contexts \mathcal{K} and the set of programs \mathcal{P} :

VII. Call-by-name solvability

Definition VII.3.13

Given a set of contexts \mathcal{K} , a set of programs \mathcal{P} , a set of inputs \mathcal{I} , a reduction \rightsquigarrow , and a set of outputs \mathcal{O} , the *observational preorder* $\sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}}$ is defined by

$$t_1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2 \stackrel{\text{def}}{=} t_1 \models t_2 \text{ and } \forall \mathbb{R} \in \mathcal{K}, \begin{matrix} \mathbb{R}[t_1] \in \mathcal{P} \\ \mathbb{R}[t_2] \in \mathcal{P} \end{matrix} \Rightarrow \mathbb{R}[t_1] \lesssim_{\mathcal{I}, \rightsquigarrow, \mathcal{O}} \mathbb{R}[t_2]$$

When \mathcal{K} is instead of a set of disubstitutions, we use the same definition on states^a

$$q_1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} q_2 \stackrel{\text{def}}{=} \forall \mathbb{R} \in \mathcal{K}, \begin{matrix} \mathbb{R}[q_1] \in \mathcal{P} \\ \mathbb{R}[q_2] \in \mathcal{P} \end{matrix} \Rightarrow \mathbb{R}[q_1] \lesssim_{\mathcal{I}, \rightsquigarrow, \mathcal{O}} \mathbb{R}[q_2]$$

$$\text{i.e. } \forall \varphi \in \mathcal{K}, \begin{matrix} q_1[\varphi] \in \mathcal{P} \\ q_2[\varphi] \in \mathcal{P} \end{matrix} \Rightarrow q_1[\varphi] \lesssim_{\mathcal{I}, \rightsquigarrow, \mathcal{O}} q_2[\varphi]$$

and extend it to non-state expressions by:

- In λ_n^\rightarrow :

$$t_n^1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_n^2 \stackrel{\text{def}}{=} \underline{t_n^1} \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} \underline{t_n^2}$$

$$e_n^1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} e_n^2 \stackrel{\text{def}}{=} \forall x^n, \underline{e_n^1}[x^n] \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} \underline{e_n^2}[x^n]$$

- In Li_n^\rightarrow :

$$t_n^1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_n^2 \stackrel{\text{def}}{=} \langle t_n^1 | \star^n \rangle \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} \langle t_n^2 | \star^n \rangle$$

$$e_n^1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} e_n^2 \stackrel{\text{def}}{=} \forall x^n, \langle x^n | e_n^1 \rangle \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} \langle x^n | e_n^2 \rangle$$

- In L_n^\rightarrow :

$$t_n^1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_n^2 \stackrel{\text{def}}{=} \forall \alpha^n, \langle t_n^1 | \alpha^n \rangle \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} \langle t_n^2 | \alpha^n \rangle$$

$$e_n^1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} e_n^2 \stackrel{\text{def}}{=} \forall x^n, \langle x^n | e_n^1 \rangle \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} \langle x^n | e_n^2 \rangle$$

The *observational equivalence* $\approx_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}}$, and the *strict observational preorder* $\sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}}$ are respectively the equivalence relation and the strict preorder induced by the observational preorder $\sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}}$:

$$t_1 \approx_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2 \stackrel{\text{def}}{=} t_1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2 \text{ and } t_1 \supseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2$$

$$t_1 \sqsubset_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2 \stackrel{\text{def}}{=} t_1 \sqsubseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2 \text{ and } t_1 \not\supseteq_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}^{\mathcal{K}, \mathcal{P}} t_2$$

To keep the subscripts short, we use the same notations as for $\lesssim_{\mathcal{I}, \rightsquigarrow, \mathcal{O}}$, and in λ_N^\rightarrow (resp. λ_n^\rightarrow , Li_n^\rightarrow or L_n^\rightarrow), we write \mathbf{K} for \mathbf{K}_N (resp. \mathbf{k}_n), and φ for φ_N (resp. φ_n). When both \mathcal{I} and \mathcal{O} are trivial, we sometimes leave them implicit: $\sqsubseteq_{\rightsquigarrow}^{\mathcal{K}, \mathcal{P}} = \sqsubseteq_{\cdot, \rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$.

^aSince $q_1 \models q_2$ always holds, we remove it from the definition.

VII. Call-by-name solvability

Fact VII.3.14

We have

$$\sqsubseteq_{\rightsquigarrow} = \sqsubseteq_{\rightsquigarrow}^{\varphi, \mathbb{Q}} = \sqsubseteq_{\rightsquigarrow, \cdot}^{\varphi, \mathbb{Q}}.$$

Proof

By definition.

As will be shown in [△](#), all definitions that use the same reduction \rightsquigarrow are actually equivalent.

Properties

As one might expect, $\sqsubseteq_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$ is a precongruence and $\widetilde{\sqsubseteq}_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$ and is congruence. For some choices of parameters, this follows trivially from the definitions, as shown below, but for others it does not. Fortunately, since all instances of $\sqsubseteq_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$ we consider are equivalent ([△](#)), it suffices to show each property for one instance, and it carries over to all the others.

Transitivity While reflexivity and symmetry always holds trivially, transitivity only holds trivially for $\mathcal{P} = \mathbb{Q}$ (but not for $\mathcal{P} = \bar{\mathbb{Q}}$, see [△](#)):

Fact VII.3.15

For $\mathcal{P} = \mathbb{Q}$, the observational preorder $\sqsubseteq_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$ is a preorder and the observational equivalence $\widetilde{\sqsubseteq}_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$ is an equivalence.

Proof

By definition of $\widetilde{\sqsubseteq}_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$, it suffices to show that $\sqsubseteq_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}}$ is a preorder. Reflexivity trivially follows from reflexivity of $\lesssim_{\rightsquigarrow, \cdot}$ (even for $\mathcal{P} \neq \mathbb{Q}$), so that we only need to show transitivity. Let t_1, t_2 , and t_3 be three terms such that

$$t_1 \sqsubseteq_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}} t_2 \sqsubseteq_{\rightsquigarrow, \cdot}^{\mathcal{K}, \mathcal{P}} t_3$$

and let \mathcal{C} be a context such that

$$\mathcal{C}[t_1] \in \mathcal{P} \quad \text{and} \quad \mathcal{C}[t_3] \in \mathcal{P}$$

We in particular have

$$t_1 \equiv t_2 \equiv t_3$$

which implies that $t_1 \equiv t_3$, and also that

$$\mathcal{C}[t_1] \in \mathbb{Q} \Rightarrow \mathcal{C}[t_2] \in \mathbb{Q} \Leftarrow \mathcal{C}[t_3] \in \mathbb{Q}$$

VII. Call-by-name solvability

Now since $\mathcal{P} = \mathcal{Q}$, we therefore have $\llbracket t_2 \rrbracket \in \mathcal{P}$, which allows us to use the two $\sqsubseteq_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}}$ hypotheses to get

$$\llbracket t_1 \rrbracket \lesssim_{\mathcal{F}, \rightsquigarrow, \emptyset} \llbracket t_2 \rrbracket \lesssim_{\mathcal{F}, \rightsquigarrow, \emptyset} \llbracket t_3 \rrbracket$$

We can therefore conclude that

$$t_1 \sqsubseteq_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}} t_3$$

Remark VII.3.16

As explained in [△](#), it is not always immediate that $\sqsubseteq_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}}$, $\sim_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}}$, and $\sqsubset_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}}$ are transitive. More precisely, transitivity is trivial when $\mathcal{P} = \mathcal{Q}$, but not when $\mathcal{P} = \mathcal{Q}$ because there is not immediate way of proving

$$\llbracket t_1 \rrbracket \in \mathcal{P} \text{ and } \llbracket t_3 \rrbracket \in \mathcal{P} \Rightarrow \llbracket t_2 \rrbracket \in \mathcal{P}$$

which is required to go from

$$t_1 \sqsubseteq_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}} t_2 \sqsubseteq_{\mathcal{F}, \rightsquigarrow, \emptyset}^{\mathcal{K}, \mathcal{P}} t_3 \text{ to } \llbracket t_1 \rrbracket \lesssim_{\mathcal{F}, \rightsquigarrow, \emptyset} \llbracket t_2 \rrbracket \lesssim_{\mathcal{F}, \rightsquigarrow, \emptyset} \llbracket t_3 \rrbracket$$

(but of course the implication holds whenever $\text{FV}_V(t_2) \subseteq \text{FV}_V(t_1) \cup \text{FV}_V(t_3)$ so that transitivity holds when we restrict these relations to terms with the same value variables).

Monotonicity Monotonicity is a very important property that states that contexts are $\sqsubseteq_{\rightsquigarrow}^{\mathcal{K}, \mathcal{P}}$ -monotone. Monotonicity for $\sqsubseteq_{\rightsquigarrow}^{\mathcal{K}, \mathcal{P}}$ is trivial:

Fact VII.3.17: Monotonicity with respect to $\sqsubseteq_{\rightsquigarrow}^{\mathcal{K}, \mathcal{P}}$

For any context $\llbracket \cdot \rrbracket$ and terms t_1 and t_2 pluggable in $\llbracket \cdot \rrbracket$, we have

$$t_1 \sqsubseteq_{\rightsquigarrow}^{\mathcal{K}_N, \mathcal{P}} t_2 \Rightarrow \llbracket t_1 \rrbracket \sqsubseteq_{\rightsquigarrow}^{\mathcal{K}_N, \mathcal{P}} \llbracket t_2 \rrbracket$$

Proof

Let $\mathcal{K} = \mathcal{K}_N$ or \mathcal{K}_n . We have

$$\begin{aligned} t_1 \sqsubseteq_{\rightsquigarrow}^{\mathcal{K}, \mathcal{P}} t_2 &\Leftrightarrow \forall \llbracket \cdot \rrbracket \in \mathcal{K}, \left. \begin{array}{l} \llbracket t_1 \rrbracket \in \mathcal{P} \\ \llbracket t_2 \rrbracket \in \mathcal{P} \end{array} \right\} \Rightarrow \llbracket t_1 \rrbracket \lesssim_{\rightsquigarrow} \llbracket t_2 \rrbracket \\ &\Leftrightarrow \forall \llbracket \cdot \rrbracket_{\blacktriangledown} \in \mathcal{K}, \forall \llbracket \cdot \rrbracket_{\blacktriangle} \in \mathcal{K}, \left. \begin{array}{l} \llbracket \llbracket \cdot \rrbracket_{\blacktriangledown} t_1 \rrbracket \in \mathcal{P} \\ \llbracket \llbracket \cdot \rrbracket_{\blacktriangle} t_2 \rrbracket \in \mathcal{P} \end{array} \right\} \Rightarrow \llbracket \llbracket \cdot \rrbracket_{\blacktriangledown} t_1 \rrbracket \lesssim_{\rightsquigarrow} \llbracket \llbracket \cdot \rrbracket_{\blacktriangle} t_2 \rrbracket \\ &\Leftrightarrow \forall \llbracket \cdot \rrbracket_{\blacktriangledown} \in \mathcal{K}, \llbracket \llbracket \cdot \rrbracket_{\blacktriangledown} t_1 \rrbracket \sqsubseteq_{\rightsquigarrow}^{\mathcal{K}, \mathcal{P}} \llbracket \llbracket \cdot \rrbracket_{\blacktriangledown} t_2 \rrbracket \end{aligned}$$

VII. Call-by-name solvability

where in the second equivalence follows from the fact that $(\mathcal{R}_\blacktriangle, \mathcal{R}_\blacktriangledown) \mapsto \mathcal{R}_\blacktriangle \boxed{\mathcal{R}_\blacktriangledown}$ is surjective and that

$$\mathcal{R} = \mathcal{R}_\blacktriangle \boxed{\mathcal{R}_\blacktriangledown} \Rightarrow \mathcal{R} t_i = (\mathcal{R}_\blacktriangle \boxed{\mathcal{R}_\blacktriangledown}) t_i = \mathcal{R}_\blacktriangle \boxed{\mathcal{R}_\blacktriangledown t_i}$$

Both immediately follow from from $(\mathcal{K}, \square, (\mathcal{R}_\blacktriangle, \mathcal{R}_\blacktriangledown) \mapsto \mathcal{R}_\blacktriangle \boxed{\mathcal{R}_\blacktriangledown})$ being a monoid (resp. non-symmetric colored operad) that acts on \mathcal{T}_{erm} via $(\mathcal{R}, t) \mapsto \mathcal{R} \boxed{t}$.

Soundness of β -conversion When comparing expressions with respect to $\stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty$, it is sound to reason module β -equivalence:

Lemma VII.3.18

For any terms t_1 and t_2 , we have

$$t_1 \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty t_2 \Leftrightarrow t_1 \approx_\beta \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty \approx_\beta t_2$$

Proof

The \Rightarrow implication is trivial by reflexivity of \approx_β . We now prove the \Leftarrow implication. Suppose that

$$t_1 \approx_\beta t'_1 \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty t'_2 \approx_\beta t_2, \quad \mathcal{R} \boxed{t_1} \in \mathcal{P}, \quad \mathcal{R} \boxed{t_2} \in \mathcal{P}$$

It is immediate that $t_1 \equiv t_2$ (because $\approx_\beta \subseteq \equiv$ by induction on the derivation).



In particular, \approx_β is sound with respect to $\stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty$:

Lemma VII.3.19

For any terms t_1 and t_2 , we have

$$t_1 \approx_\beta t_2 \Rightarrow t_1 \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty t_2$$

Proof

By transitivity of \approx_β , reflexivity of $\stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty$, and the previous lemma, we have

$$t_1 \approx_\beta t_2 \Rightarrow t_1 \approx_\beta \approx_\beta t_2 \Rightarrow t_1 \approx_\beta \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty \approx_\beta t_2 \Rightarrow t_1 \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq}_\infty t_2$$

VII. Call-by-name solvability

Remark VII.3.20

Note that when $\stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq_{\infty}}$ is transitive, we also have

$$t_1 \approx_{\beta} \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq_{\infty}} t_2 \Rightarrow t_1 \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq_{\infty}} \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq_{\infty}} \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq_{\infty}} t_2 \Rightarrow t_1 \stackrel{\mathcal{K}, \mathcal{P}}{\sqsubseteq_{\infty}} t_2$$

so that the last two lemmas are actually equivalent.

VII.3.4. Operational relevance and solvability



VII.4. Equivalences between definitions

VIII. Call-by-value solvability



IX. Polarized solvability



Bibliography

- [Abr90] S. Abramsky, “The lazy lambda calculus,” 1990 (cit. on pp. 5, 9, 161, 184).
- [AbrOng93] S. Abramsky and C.-H. L. Ong, “Full abstraction in the lazy lambda calculus,” *Inf. Comput.*, vol. 105, no. 2, pp. 159–267, Aug. 1993, ISSN: 0890-5401. DOI: [10.1006/inco.1993.1044](https://doi.org/10.1006/inco.1993.1044). [Online]. Available: <https://doi.org/10.1006/inco.1993.1044> (cit. on p. 8).
- [AccGue16] B. Accattoli and G. Guerrieri, “Open call-by-value,” in *Programming Languages and Systems*, A. Igarashi, Ed., Cham: Springer International Publishing, 2016, pp. 206–226, ISBN: 978-3-319-47958-3 (cit. on pp. 5, 23, 76).
- [AccPao12] B. Accattoli and L. Paolini, “Call-by-value solvability, revisited,” in *Functional and Logic Programming*, T. Schrijvers and P. Thiemann, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 4–16, ISBN: 978-3-642-29822-6 (cit. on pp. 5, 7, 23).
- [Bar84] H. Barendregt, *The lambda calculus: its syntax and semantics* (Studies in logic and the foundations of mathematics). North-Holland, 1984, ISBN: 9780444867483. [Online]. Available: <https://books.google.fr/books?id=eMtTAAAYAAJ> (cit. on pp. 3, 5, 9, 18, 20, 76, 117, 185, 196, 233).
- [BucKesRíoVis20] A. Bucciarelli, D. Kesner, A. Ríos, and A. Viso, “The bang calculus revisited,” in *Functional and Logic Programming*, K. Nakano and K. Sagonias, Eds., Cham: Springer International Publishing, 2020, pp. 13–32, ISBN: 978-3-030-59025-3 (cit. on pp. 5, 76).
- [Chu85] A. Church, *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. USA: Princeton University Press, 1985, ISBN: 0691083940 (cit. on p. 8).
- [CurFioMun16] P.-L. Curien, M. Fiore, and G. Munch-Maccagnoni, “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi,” in *Proc. POPL*, 2016. DOI: [10.1145/2837614.2837652](https://doi.org/10.1145/2837614.2837652) (cit. on pp. 6, 76, 81, 84, 91, 123).

Bibliography

- [CurHer00] P.-L. Curien and H. Herbelin, “The duality of computation,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00)*, Montreal, Canada, September 18-21, 2000, ser. SIGPLAN Notices 35(9), ACM, 2000, pp. 233–243, ISBN: 1-58113-202-6. DOI: <http://doi.acm.org/10.1145/351240.351262> (cit. on pp. 4–6, 16, 18, 62).
- [CurMun10] P.-L. Curien and G. Munch-Maccagnoni, “The duality of computation under focus,” in *IFIP TCS*, C. S. Calude and V. Sassone, Eds., ser. IFIP Advances in Information and Communication Technology, vol. 323, Springer, 2010, pp. 165–181 (cit. on p. 6).
- [DanNie04] O. Danvy and L. R. Nielsen, “Refocusing in reduction semantics,” *BRICS Report Series*, vol. 11, no. 26, Nov. 2004. DOI: [10.7146/brics.v11i26.21851](https://doi.org/10.7146/brics.v11i26.21851). [Online]. Available: <https://tidsskrift.dk/brics/article/view/21851> (cit. on pp. 18, 29, 49).
- [dVri16] F.-J. de Vries, “On Undefined and Meaningless in Lambda Definability,” in *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, D. Kesner and B. Pientka, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 52, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 18:1–18:15, ISBN: 978-3-95977-010-1. DOI: [10.4230/LIPIcs.FSCD.2016.18](https://doi.org/10.4230/LIPIcs.FSCD.2016.18). [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2016/5978> (cit. on p. 7).
- [DezGio01] M. Dezani-Ciancaglini and E. Giovannetti, “From Böhm’s Theorem to Observational Equivalences: an Informal Account,” in *BOTH’01*, ser. Electronic Notes in Theoretical Computer Science (<http://www.elsevier.nl/locate/entcs/volume50>), vol. 50, Elsevier, 2001, pp. 83–116. [Online]. Available: <http://www.di.unito.it/~dezani/papers/both01.ps> (cit. on pp. 9, 184, 185).
- [Dij68] E. W. Dijkstra, “Letters to the editor: Go to statement considered harmful,” *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968, ISSN: 0001-0782. DOI: [10.1145/362929.362947](https://doi.org/10.1145/362929.362947). [Online]. Available: <https://doi.org/10.1145/362929.362947> (cit. on p. 3).
- [DowAri18] P. Downen and Z. M. Ariola, “A tutorial on computational classical logic and the sequent calculus,” *Journal of Functional Programming*, vol. 28, e3, 2018. DOI: [10.1017/S0956796818000023](https://doi.org/10.1017/S0956796818000023) (cit. on p. 6).
- [Ehr16] T. Ehrhard, “Call-by-push-value from a linear logic point of view,” in *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, Berlin, Heidelberg: Springer-Verlag, 2016, pp. 202–228, ISBN: 9783662494974 (cit. on p. 76).

Bibliography

- [EhrGue16] T. Ehrhard and G. Guerrieri, “The bang calculus: An untyped lambda-calculus generalizing call-by-name and call-by-value,” in *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP’16, Edinburgh, United Kingdom: Association for Computing Machinery, 2016, pp. 174–187, ISBN: 9781450341486. DOI: [10.1145/2967973.2968608](https://doi.org/10.1145/2967973.2968608). [Online]. Available: <https://doi.org/10.1145/2967973.2968608> (cit. on pp. 5, 76).
- [ErwRen04] M. Erwig and D. Ren, “Monadification of functional programs,” *Sci. Comput. Program.*, vol. 52, no. 1–3, pp. 101–129, Aug. 2004, ISSN: 0167-6423. DOI: [10.1016/j.scico.2004.03.004](https://doi.org/10.1016/j.scico.2004.03.004). [Online]. Available: <https://doi.org/10.1016/j.scico.2004.03.004> (cit. on p. 84).
- [GarNog16] Á. García-Pérez and P. Nogueira, “No solvable lambda-value term left behind,” *Logical Methods in Computer Science*, vol. Volume 12, Issue 2, Jun. 2016. DOI: [10.2168/LMCS-12\(2:12\)2016](https://lmcs.episciences.org/1644). [Online]. Available: <https://lmcs.episciences.org/1644> (cit. on p. 5).
- [Gir11] J.-Y. Girard, “The blind spot: Lectures on logic,” 2011 (cit. on p. 134).
- [Hue97] G. P. Huet, “The zipper,” *J. Funct. Program.*, vol. 7, no. 5, pp. 549–554, 1997. DOI: [10.1017/s0956796897002864](https://doi.org/10.1017/s0956796897002864). [Online]. Available: <https://doi.org/10.1017/s0956796897002864> (cit. on p. 30).
- [IntManPol17] B. Intrigila, G. Manzonetto, and A. Polonsky, “Refutation of sallé’s long-standing conjecture,” in *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, D. Miller, Ed., ser. LIPIcs, vol. 84, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 20:1–20:18. DOI: [10.4230/LIPIcs.FSCD.2017.20](https://doi.org/10.4230/LIPIcs.FSCD.2017.20). [Online]. Available: <https://doi.org/10.4230/LIPIcs.FSCD.2017.20> (cit. on pp. 9, 185).
- [Kri07] J.-L. Krivine, “A call-by-name lambda-calculus machine,” *Higher Order Symbolic Computation*, vol. 20, pp. 199–207, 2007. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-00154508> (cit. on pp. 6, 16, 18, 30).
- [Lev01] P. B. Levy, “Call-by-push-value,” Ph.D. dissertation, Queen Mary University of London, UK, 2001. [Online]. Available: <http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233> (cit. on p. 91).
- [Lev04] P. B. Levy, *Call-By-Push-Value: A Functional/Imperative Synthesis* (Semantics Structures in Computation). Springer, 2004, vol. 2, ISBN: 1-4020-1730-8 (cit. on pp. 4, 5, 76, 91).
- [Lev06] P. B. Levy, “Call-by-push-value: Decomposing call-by-value and call-by-name,” *High. Order Symb. Comput.*, vol. 19, no. 4, pp. 377–414, 2006. DOI: [10.1007/s10990-006-0480-6](https://doi.org/10.1007/s10990-006-0480-6). [Online]. Available: <https://doi.org/10.1007/s10990-006-0480-6> (cit. on pp. 4, 5, 76, 91).

Bibliography

- [Mog89] E. Moggi, “Computational lambda-calculus and monads,” in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS ’89)*, Pacific Grove, California, USA, June 5-8, 1989, IEEE Computer Society, 1989, pp. 14–23. DOI: [10 . 1109 / LICS . 1989 . 39155](https://doi.org/10.1109/LICS.1989.39155). [Online]. Available: <https://doi.org/10.1109/LICS.1989.39155> (cit. on pp. 5, 91).
- [Mog91] E. Moggi, “Notions of computation and monads,” *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, 1991. DOI: [10 . 1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4). [Online]. Available: [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4) (cit. on p. 5).
- [Mor69] J. H. Morris, “Lambda calculus models of programming languages,” Ph.D. dissertation, Massachusetts Institute of Technology, 1969 (cit. on pp. 9, 184, 187).
- [Mun13] G. Munch-Maccagnoni, “Syntax and Models of a non-Associative Composition of Programs and Proofs,” Ph.D. dissertation, Univ. Paris Diderot, 2013 (cit. on p. 84).
- [Mun14] G. Munch-Maccagnoni, “Models of a Non-Associative Composition,” in *Proceedings of the 17th International Conference on Foundations of Software Science and Computation Structures (FoSSaCs)*, A. Muscholl, Ed., ser. Lecture Notes in Computer Science, vol. 8412, Springer Heidelberg, 2014, pp. 397–412 (cit. on p. 84).
- [MunSch15] G. Munch-Maccagnoni and G. Scherer, “Polarised Intermediate Representation of Lambda Calculus with Sums,” in *Proceedings of the Thirtieth Annual ACM/IEEE Symposium on Logic In Computer Science (LICS 2015)*, 2015. DOI: [10 . 1109/LICS . 2015 . 22](https://doi.org/10.1109/LICS.2015.22) (cit. on pp. 6, 76).
- [Ong88] C. L. Ong, “Fully abstract models of the lazy lambda calculus,” in *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, IEEE Computer Society, 1988, pp. 368–376. DOI: [10 . 1109 / SFCS . 1988 . 21953](https://doi.org/10.1109/SFCS.1988.21953). [Online]. Available: <https://doi.org/10.1109/SFCS.1988.21953> (cit. on p. 5).
- [PaoRon99] L. Paolini and S. Ronchi Della Rocca, “Call-by-value solvability,” *RAIRO Theor. Informatics Appl.*, vol. 33, no. 6, pp. 507–534, 1999. DOI: [10 . 1051 / ita : 1999130](https://doi.org/10.1051/ita:1999130). [Online]. Available: <https://doi.org/10.1051/ita:1999130> (cit. on pp. 5, 23).
- [Par92] M. Parigot, “ $\lambda\mu$ -calculus: An algorithmic interpretation of classical natural deduction,” in *Logic Programming and Automated Reasoning*, A. Voronkov, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 190–201, ISBN: 978-3-540-47279-7 (cit. on p. 5).

Bibliography

- [Reg94] L. Regnier, “Une équivalence sur les lambda-termes,” *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 281–292, 1994. DOI: [10.1016/0304-3975\(94\)90012-4](https://doi.org/10.1016/0304-3975(94)90012-4). [Online]. Available: [https://doi.org/10.1016/0304-3975\(94\)90012-4](https://doi.org/10.1016/0304-3975(94)90012-4) (cit. on pp. 16, 18, 23).
- [Tak95] M. Takahashi, “Parallel reductions in λ -calculus,” *Inf. Comput.*, vol. 118, no. 1, pp. 120–127, Apr. 1995, ISSN: 0890-5401. DOI: [10.1006/inco.1995.1057](https://doi.org/10.1006/inco.1995.1057). [Online]. Available: <https://doi.org/10.1006/inco.1995.1057> (cit. on pp. 117, 233).
- [Wad76] C. P. Wadsworth, “The relation between computational and denotational properties for scott’s d_{infty} -models of the lambda-calculus,” *SIAM J. Comput.*, vol. 5, no. 3, pp. 488–521, 1976. DOI: [10.1137/0205036](https://doi.org/10.1137/0205036). [Online]. Available: <https://doi.org/10.1137/0205036> (cit. on pp. 7, 9, 184).

Appendix



.1. Properties of disubstitutions

Recall the definitions of disubstitutions in the different calculi:

Summary .1.1

- In λ -calculi, a disubstitution φ is a pair $\varphi = (\sigma, \mathbb{S})$ that consists of a substitution σ and a stack \mathbb{S} , with

$$T[\varphi] = \mathbb{S}[T[\sigma]]$$

- In λ -calculi with focus, a disubstitution φ is a pair $\varphi = (\sigma, \mathbb{S})$ that consists of a substitution σ and a stack \mathbb{S} , with

$$t[\varphi] \stackrel{\text{def}}{=} t[\sigma]$$

$$c[\varphi] \stackrel{\text{def}}{=} \text{defer}(c[\sigma], \mathbb{S})$$

$$e[\varphi] \stackrel{\text{def}}{=} \text{defer}(e[\sigma], \mathbb{S})$$

- In L-calculi, a disubstitution φ is a substitution whose domain may contain stack variables α in addition to the usual value variable x .

We now define:

Definition .1.2


In each calculus, we define

$$\varphi_2 \circ \varphi_1 \stackrel{\text{def}}{=} \varphi_1[\varphi_2] \quad \text{and} \quad \varphi \bullet t \stackrel{\text{def}}{=} t[\varphi]$$

We also define

$$1_{\circ} \stackrel{\text{def}}{=} (\text{Id}_{\mathcal{V}}, \square) \quad (\text{resp. } 1_{\circ} \stackrel{\text{def}}{=} \text{Id}_{\mathcal{V} \cup \mathcal{S}})$$

Fact .1.3

In each calculus, the set of disubstitutions φ has a monoid structure $(\varphi, \circ, 1_{\circ})$ and this monoid acts on commands, expressions,  and evaluation contexts via \bullet .

Proof



While substitutions that act on both value variables x^n and the stack variable \star^n really are substitutions, we call them disubstitutions to avoid any confusion:

Definition .1.4

We call *disubstitutions*, and denote by φ , that act on both value variables and stack variables.

Since we only have one stack variable in Li_n^\rightarrow , those are of the shape $\sigma, \star^n \mapsto s_n$. The action of disubstitutions on terms, and their compositions are defined in the expected way. A full description of their action can be found in the right column of Figure ??.

Since terms are either variable x^n , or bind \star^n , only having one stack variable \star^n enforces the following property:

Fact .1.5

Term t_n have no free stack variables, i.e.

$$\text{FV}_s(t_n) = \emptyset$$

Command c_n and evaluation contexts e_n have exactly one free stack variable \star^n , i.e.

$$\text{FV}_s(c_n) = \text{FV}_s(e_n) = \{\star^n\}$$

Proof

By induction.

Terms having no free stack variables implies disubstitutions can be decomposed as a substitution and a disubstitution of the shape $\star^n \mapsto s_n$:

Fact .1.6

Given a disubstitution $\varphi = \sigma, \star^n \mapsto s_n$:

- for any expression, evaluation context or command t ,

$$t[\sigma, s_n / \star^n] = t[\sigma][s_n / \star^n]$$

- for any expression t_n ,

$$t_n[\sigma, s_n / \star^n] = t_n[\sigma]$$

Proof

- $t_n[\sigma, s_n / \star^n] = t_n[\sigma]$ By induction on t_n .
- $t[\sigma, s_n / \star^n] = t[\sigma][s_n / \star^n]$ By induction on t , using the fact that $\sigma[s_n / \star^n] = \sigma$ by the previous bullet.

This also allows simplifying the composition of two disubstitutions:

Fact .1.7

For any disubstitutions $\varphi_1 = \sigma_1, \star^n \mapsto s_n^1$ and $\varphi_2 = \sigma_2, \star^n \mapsto s_n^2$, we have

$$\varphi_1[\varphi_2] = \sigma_1[\sigma_2], \star^n \mapsto s_n^1[s_n^2[\sigma_1]/\star^n]$$

Proof

By the previous fact,

$$\varphi_1[\varphi_2] = (\sigma_1[\varphi_2], \star^n \mapsto s_n^1[\varphi_2])\sigma_1[\sigma_2], \star^n \mapsto s_n^1[s_n^2[\sigma_1]/\star^n]$$

.2. Properties of reductions



.3. Detailed proofs

Fact A.1.8: Equivalence between \triangleright^\oplus and \boxtimes^\oplus

- The \boxtimes -normal expressions are exactly the \triangleright -normal expressions:

$$T_N \boxtimes \Leftrightarrow T_N \triangleright$$

- The \boxtimes steps can be postponed at the cost of strengthening $\triangleright_{\text{let}}$ to $\triangleright_{\text{let}}$:

$$T_N \boxtimes^* T'_N \Leftrightarrow T_N \triangleright^* \boxtimes^* T'_N$$

- Evaluating with \boxtimes or \triangleright yields the same result:

$$T_N \boxtimes^\oplus T'_N \Leftrightarrow T_N \triangleright^\oplus T'_N$$

Proof of Fact I.1.8 from page 25

Recall that

$$\triangleright = \triangleright_{\rightarrow} \cup \triangleright_{\text{let}} \quad \text{and} \quad \boxtimes = \triangleright_{\rightarrow} \cup \triangleright_{\text{let}} \cup \boxtimes$$

- $T_N \triangleright \Leftrightarrow T_N \boxtimes$ Take \mathcal{S}_N maximal such that $T_N = \mathcal{S}_N \boxed{U_N}$. The result is immediate by case analysis on \mathcal{S}_N and U_N .

- $T_N \boxtimes^* T'_N \Leftrightarrow T_N \triangleright^* \boxtimes^* T'_N$ Since we have $\boxtimes \cup \triangleright_{\rightarrow} \subseteq \boxtimes$ by definition, it suffices to show that $\triangleright_{\text{let}} \subseteq \boxtimes^*$. This is immediate: any reduction

$$(\text{let } x^N := T_N \text{ in } U_N) V_N^1 \dots V_N^q \triangleright_{\text{let}} (U_N[T_N/x^N]) V_N^1 \dots V_N^q$$

can be simulated by

$$\begin{aligned} (\text{let } x^N := T_N \text{ in } U_N) V_N^1 \dots V_N^q &\boxtimes (\text{let } x^N := T_N \text{ in } U_N V_N^1) V_N^2 \dots V_N^q \\ &\boxtimes^* \text{let } x^N := T_N \text{ in } U_N V_N^1 \dots V_N^q \\ &\triangleright_{\text{let}} (U_N[T_N/x^N]) V_N^1 \dots V_N^q \end{aligned}$$

- $T_N \boxtimes^* \triangleright T'_N \Rightarrow T_N \triangleright T'_N$ By induction on the number of \boxtimes steps and case analysis on T_N .

- $T_N \boxtimes^* T'_N \Rightarrow T_N \triangleright^* \boxtimes^* T'_N$ Suppose that $T_N \boxtimes^* T'_N$. By definition of \boxtimes (and monotonicity of the reflexive transitive closure), we have $T_N (\triangleright \cup \boxtimes)^* T'_N$. By the previous bullet, this simplifies to $T_N \triangleright^* \boxtimes^* T'_N$.

- $T_N \boxtimes^\oplus T'_N \Leftrightarrow T_N \triangleright^\oplus T'_N$ The \Leftarrow implication follows from the previous bullets. Now suppose that $T_N \boxtimes^\oplus T'_N$. By the previous bullets, we have $T_N \triangleright^* \boxtimes^l T'_N \triangleright$ for some l . Since any \boxtimes -reduct is \triangleright -reducible, we necessarily have $l = 0$, and we are done.

Fact A.4.3

In $\vec{\lambda}_N$ (resp. M_N^{\rightarrow}), the set of stacks \mathbf{S}_N has a monoid structure

$$(\mathbf{S}_N, \circ_{\square}, \square) \quad (\text{resp. } (\mathbf{S}_N, \circ_{\star}, \star^N))$$

where

$$\mathbb{S}_N^2 \circ_{\square} \mathbb{S}_N^1 \stackrel{\text{def}}{=} \mathbb{S}_N^2 \boxed{\mathbb{S}_N^1} \quad (\text{resp. } S_N^1 \circ_{\star} S_N^2 \stackrel{\text{def}}{=} S_N^1[S_N^2/\star^N])$$

and this monoid acts on configurations on the left (resp. on the right) via

$$\mathbb{S}_N \bullet_{\square} C_N \stackrel{\text{def}}{=} \mathbb{S}_N \boxed{C_N} \quad (\text{resp. } C_N \bullet_{\star} S_N \stackrel{\text{def}}{=} C_N[S_N/\star^N])$$

In other words:

- (mon-unit) for any stack \mathbb{S}_N (resp. S_N), we have

$$\square \boxed{\mathbb{S}_N} = \mathbb{S}_N = \mathbb{S}_N \square \quad (\text{resp. } \star^N[S_N/\star^N] = S_N = S_N[\star^N/\star^N])$$

- (mon-accoc) for any stacks $\mathbb{S}_N^1, \mathbb{S}_N^2$, and \mathbb{S}_N^3 (resp. S_N^1, S_N^2 , and S_N^3), we have

$$\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2 \boxed{\mathbb{S}_N^1}} = (\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2}) \boxed{\mathbb{S}_N^1} \quad (\text{resp. } S_N^1[S_N^2/\star^N][S_N^3/\star^N] = S_N^1[S_N^2[S_N^3/\star^N]/\star^N])$$

- (act-unit) for any configuration C_N , we have

$$\square \boxed{C_N} = C_N \quad (\text{resp. } C_N = C_N[\star^N/\star^N])$$

- (act-assoc) for any configuration C_N^1 and stacks \mathbb{S}_N^2 and \mathbb{S}_N^3 (resp. S_N^2 and S_N^3), we have

$$\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2 \boxed{C_N^1}} = (\mathbb{S}_N^3 \boxed{\mathbb{S}_N^2}) \boxed{C_N^1} \quad (\text{resp. } C_N^1[S_N^2/\star^N][S_N^3/\star^N] = C_N^1[S_N^2[S_N^3/\star^N]/\star^N])$$

Proof of Fact I.4.3 from page 38

- (mon-unit) One equality is by definition and the other is by induction on \mathbb{S}_N (resp. S_N).
- (mon-accoc) By induction on the size of \mathbb{S}_N^2 (resp. S_N^2). The base case $\mathbb{S}_N^2 = \square$ (resp. $S_N^2 = \star^N$), follows from (i). The inductive case follows from several applications of the induction hypothesis.
- (act-unit) By definition (resp. by induction on C_N).
- (act-assoc) By induction on \mathbb{S}_N^2 (resp. S_N^2). The base case $\mathbb{S}_N^2 = \square$ (resp. $S_N^2 = \star^N$) follows from (i) and (iii). The inductive case follows from several applications of the induction hypothesis.

Fact E.2.8

The following are equivalent:

- (i) there exists a derivation of well-polarization which is valid in $\text{Li}_p^{\bar{\tau}}$ but not in $\text{Lm}_p^{\bar{\tau}}$;
- (ii) there exists a derivation of well-polarization which is valid in $\text{Li}_p^{\bar{\tau}}$ but not in $\text{Lm}_p^{\bar{\tau}}$, and whose conclusion is of the shape

$$c : (\Gamma \vdash) \quad \text{or} \quad \Gamma \mid \underline{s_\varepsilon} : \varepsilon \vdash$$

i.e. has no succedent;

- (iii) there exists a stack s_ε in $\text{Li}_p^{\bar{\tau}}$ such that $\Gamma \mid \underline{s_\varepsilon} : \varepsilon \vdash$ is derivable for some Γ ;
- (iv) at least one of the following holds:
 - (a) there exists a stack constructor $\mathfrak{s}_k^{\tau_j}$ with zero stack arguments (e.g. $\neg_-(v_+)$ or $\tilde{()}$); or
 - (b) there exists a positive type former τ_+^j whose value constructors $\mathfrak{v}_k^{\tau_+^j}$ all have exactly one stack arguments (e.g. \neg_+ or 0).
- (v) there exists a stack s_ε in $\text{Li}_p^{\bar{\tau}}$ of the shape

$$s_\varepsilon = \mathfrak{s}_k^{\tau_j}(\vec{x}) \quad (\text{e.g. } \neg_-(x^+) \quad \text{or} \quad \tilde{()})$$

or

$$s_\varepsilon = \tilde{\mu} \left[\mathfrak{v}_1^{\tau_+^j}(\vec{x}_1, \alpha_1^{\varepsilon_1}, \vec{y}_1) \cdot \langle z_1^{\varepsilon_1} \mid \alpha_1^{\varepsilon_1} \rangle^{\varepsilon_1} \right. \\ \vdots \\ \left. \mathfrak{v}_l^{\tau_+^j}(\vec{x}_l, \alpha_l^{\varepsilon_l}, \vec{y}_l) \cdot \langle z_l^{\varepsilon_l} \mid \alpha_l^{\varepsilon_l} \rangle^{\varepsilon_l} \right] \quad (\text{e.g. } \tilde{\mu}_{\neg_+}(\alpha^-) \cdot \alpha \langle x^- \mid \alpha^- \rangle^- \quad \text{or} \quad \tilde{\mu}[])$$

Furthermore, if all positive type formers in $\bar{\tau}$ have at least one constructor (i.e. there are no copies of 0), then these are also equivalent to:

- (vi) $\text{Lm}_p^{\bar{\tau}} \subsetneq \text{Li}_p^{\bar{\tau}}$.

In particular, for $\bar{\tau} \subseteq \{\rightarrow, \downarrow, \uparrow, \neg_-, \neg_+, \otimes, \wp, \oplus, \&, 1, \perp, \top\}^a$, we have

$$\text{Lm}_p^{\bar{\tau}} \subsetneq \text{Li}_p^{\bar{\tau}} \quad \Leftrightarrow \quad \bar{\tau} \cap \{\neg_-, \neg_+, \perp\} \neq \emptyset$$

^aNote the absence of 0 .

- $(i) \Rightarrow (ii)$ This derivation necessarily uses a sequent of the shape

$$c : (\Gamma \vdash) \quad \text{or} \quad \Gamma \mid \underline{e_\varepsilon : \varepsilon} \vdash$$

and there is therefore a subderivation whose conclusion is that sequent.

- $(ii) \Rightarrow (iii)$ By induction on the derivation: if the derivation ends with

$$\frac{\Gamma_1 \vdash \underline{t_\varepsilon : A_\varepsilon} \mid \quad \Gamma_2 \mid \underline{e_\varepsilon : A_\varepsilon} \vdash}{\langle t_\varepsilon \mid e_\varepsilon \rangle^\varepsilon : (\Gamma_1, \Gamma_2 \vdash)} \text{ (CUT)} \quad \left(\text{resp. } \frac{c : (\Gamma, x^\varepsilon : A_\varepsilon \vdash)}{\Gamma \mid \underline{\tilde{\mu}x^\varepsilon.c : A_\varepsilon} \vdash} (\tilde{\mu}\vdash) \right)$$

then we apply the induction hypothesis to the derivation of $\Gamma_2 \mid \underline{e_\varepsilon : A_\varepsilon} \vdash$ (resp. $c : (\Gamma, x^\varepsilon : A_\varepsilon \vdash)$).

- $(iii) \Rightarrow (iv)$ By induction on the derivation. If the last rule of the derivation is

$$\frac{\begin{array}{c} \Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : A_{\varepsilon_1}^1} \mid \quad \dots \quad \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : A_{\varepsilon_q}^q} \mid \\ \Gamma_{q+1} \mid \underline{s_{\varepsilon_{q+1}}^1 : A_{\varepsilon_{q+1}}^{q+1}} \vdash \quad \dots \quad \Gamma_{q+r} \mid \underline{s_{\varepsilon_{q+r}}^r : A_{\varepsilon_{q+r}}^{q+r}} \vdash \end{array}}{\Gamma_1, \dots, \Gamma_{q+r} \mid \underline{\mathfrak{s}_k^{\tau_j}(v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1, \dots, s_{\varepsilon_{q+r}}^r) : \tau_j^-(\vec{B})} \vdash} (\mathfrak{s}_k^{\tau_j} \vdash)$$

then either $r > 0$ and we apply the induction hypothesis to one of the derivations of $\Gamma_{q+k} \mid \underline{s_{\varepsilon_{q+k}}^k : A_{\varepsilon_{q+k}}^{q+k}} \vdash$, or $r = 0$, and we can immediately conclude that (iv). If the last rule is

$$\frac{\begin{array}{c} c_1 : (\Gamma, \vec{x}_1 : \vec{A}^1 \vdash \alpha_{\varepsilon_1}^1 : B_{\varepsilon_1}^1) \quad \dots \quad c_l : (\Gamma, \vec{x}_l : \vec{A}^l \vdash \alpha_{\varepsilon_l}^l : B_{\varepsilon_l}^l) \\ \Gamma \mid \underline{\tilde{\mu}[\mathfrak{b}_1^{\tau_j^+}(\vec{x}_1, \alpha_1).c_1 \dots \mathfrak{b}_l^{\tau_j^+}(\vec{x}_l, \alpha_l).c_l] : \tau_j^+(\vec{C})} \vdash \end{array}}{\Gamma \mid \underline{\tilde{\mu}[\mathfrak{b}_1^{\tau_j^+}(\vec{x}_1, \alpha_1).c_1 \dots \mathfrak{b}_l^{\tau_j^+}(\vec{x}_l, \alpha_l).c_l] : \tau_j^+(\vec{C})} \vdash} (\tau_j^+ \vdash)$$

we can immediately conclude that (iv).

- $(iv) \Rightarrow (v)$ Immediate.
- $(v) \Rightarrow (i)$ If $s_\varepsilon \neq \tilde{\mu}[]$, then we have $s_\varepsilon \notin \text{Lm}_p^{\tilde{\tau}}$, and in particular, the derivation that shows that $s_\varepsilon \in \text{Li}_p^{\tilde{\tau}}$ works. For $s_\varepsilon = \tilde{\mu}[]$, there are two possible shapes for the derivation of $s_\varepsilon \in \text{Li}_p^{\tilde{\tau}}$

$$\frac{}{\Gamma \mid \underline{\tilde{\mu}[] : 0} \vdash \alpha^\varepsilon : A_\varepsilon} (0\vdash) \quad \text{and} \quad \frac{}{\Gamma \mid \underline{\tilde{\mu}[] : 0} \vdash} (0\vdash)$$

and while the former is also valid in $\text{Lm}_p^{\tilde{\tau}}$, the latter is not.

- $(v) \Rightarrow (iv)$ We have $s_\varepsilon \in \text{Li}_p^{\tilde{\tau}} \setminus \text{Lm}_p^{\tilde{\tau}}$. This is immediate for the case $s_\varepsilon = \mathfrak{s}_k^{\tau_j}(\vec{x})$,

and since all positive type formers have at least one constructor, the case

$$s_\epsilon = \tilde{\mu} \left[\begin{array}{c} \mathfrak{b}_1^{\tau_1^j}(\vec{x}_1, \alpha_1^{\epsilon_1}, \vec{y}_1) \cdot \langle z_1^{\epsilon_1} | \alpha_1^{\epsilon_1} \rangle^{\epsilon_1} \\ \vdots \\ \mathfrak{b}_l^{\tau_l^j}(\vec{x}_l, \alpha_l^{\epsilon_l}, \vec{y}_l) \cdot \langle z_l^{\epsilon_l} | \alpha_l^{\epsilon_l} \rangle^{\epsilon_l} \end{array} \right]$$

is restricted to $l > 0$, i.e. $s_\epsilon = \tilde{\mu}[]$ is ruled out, which ensures that $s_\epsilon \notin \text{Lm}_p^{\vec{\tau}}$.

- $\boxed{\text{(iv)} \Rightarrow \text{(i)}}$ Take any $t \in \text{Li}_p^{\vec{\tau}} \setminus \text{Lm}_p^{\vec{\tau}}$. The derivation that $t \in \text{Li}_p^{\vec{\tau}}$ works.

Fact G.1.14

In $\lambda_N^{\vec{\tau}}$, $\lambda_n^{\vec{\tau}}$, $\text{Li}_n^{\vec{\tau}}$, and $L_n^{\vec{\tau}}$, we have

$$\mathfrak{h}_{\triangleright}^{\otimes} = \mathfrak{a}_{\triangleright}^{\otimes} \quad \text{and} \quad \mathfrak{l}_{\triangleright}^{\otimes} = \mathfrak{r}_{\triangleright}^{\otimes}$$

Proof of Fact VII.1.14 from page 197

- $\boxed{\mathfrak{h}_{\triangleright}^{\otimes} = \mathfrak{a}_{\triangleright}^{\otimes}}$ The \subseteq inclusion follows from $\mathfrak{h}_{\triangleright}$ being a strategy for $\mathfrak{a}_{\triangleright}$ (Fact VII.1.13), and the \supseteq inclusion follows from $\mathfrak{a}_{\triangleright}$ having uniqueness of termination behavior (Fact VII.1.12).
- $\boxed{\mathfrak{l}_{\triangleright}^{\otimes} = \mathfrak{r}_{\triangleright}^{\otimes}}$ By Fact VII.1.13, it suffices to show the \supseteq inclusion.
 - For $\lambda_N^{\vec{\tau}}$, this is Theorem 13.2.2 page 326 of [Bar84], which is proven by using the Standardization Theorem (Theorem 11.4.7 page 300 of [Bar84]), which is in turn proven by iterating the factorization $\rightarrow^* = \mathfrak{h}_{\triangleright}^* \mathfrak{r}_{\triangleright}^*$ (Theorem 11.4.6 of [Bar84], Corollary 4.6 of [Tak95]).
 - For $L_n^{\vec{\tau}}$, the same strategy works, though we prefer using the slightly simpler factorization $\rightarrow^* = \triangleright^* \mathfrak{r}_{\triangleright}^*$ (see Proposition ?? for the proof of this factorization in $L_p^{\vec{\tau}}$).

.4. Extra figures

Figure .4.1: Well polarized $L_p^{\bar{\tau}}$

Figure .4.1.a: Core rules

$$\begin{array}{c}
 \frac{}{x^\varepsilon : \varepsilon \vdash \underline{x^\varepsilon : \varepsilon} \mid} \text{(\vdash AX)} \qquad \frac{}{\mid \underline{\alpha^\varepsilon : \varepsilon} \vdash \alpha^\varepsilon : \varepsilon} \text{(AX\vdash)} \\
 \\
 \frac{c : (\Gamma \vdash \alpha^\varepsilon : \varepsilon, \Delta)}{\Gamma \vdash \underline{\mu \alpha^\varepsilon . c : \varepsilon} \mid \Delta} \text{(\vdash } \mu \text{)} \qquad \frac{c : (\Gamma, x^\varepsilon : \varepsilon \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu} x^\varepsilon . c : \varepsilon} \vdash \Delta} \text{(\tilde{\mu} \vdash)} \\
 \\
 \frac{\Gamma_1 \vdash \underline{t_\varepsilon : \varepsilon} \mid \Delta_1 \quad \Gamma_2 \mid \underline{e_\varepsilon : \varepsilon} \vdash \Delta_2}{\langle \underline{t_\varepsilon} \mid \underline{e_\varepsilon} \rangle^\varepsilon : (\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2)} \text{(CUT)}
 \end{array}$$

Figure .4.1.b: Structural rules (commands)

$$\begin{array}{c}
 \frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma \vdash \alpha^\varepsilon : \varepsilon, \Delta)} \text{(\vdash wc)} \qquad \frac{c : (\Gamma \vdash \alpha_1^\varepsilon : \varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta)}{c[\beta^\varepsilon / \alpha_1^\varepsilon, \beta^\varepsilon / \alpha_2^\varepsilon] : (\Gamma \vdash \beta^\varepsilon : \varepsilon, \Delta)} \text{(\vdash cc)} \\
 \\
 \frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma, x^\varepsilon : \varepsilon \vdash \Delta)} \text{(wc\vdash)} \qquad \frac{c : (\Gamma, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon \vdash \Delta)}{c[y^\varepsilon / x_1^\varepsilon, y^\varepsilon / x_2^\varepsilon] : (\Gamma, y^\varepsilon : \varepsilon \vdash \Delta)} \text{(cc\vdash)} \\
 \\
 \frac{c : (\Gamma \vdash \Delta_1, \alpha_1^\varepsilon : \varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta_2)}{c : (\Gamma \vdash \Delta_1, \alpha_2^\varepsilon : \varepsilon, \alpha_1^\varepsilon : \varepsilon, \Delta_2)} \text{(\vdash pc)} \qquad \frac{c : (\Gamma_1, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon, \Gamma_2 \vdash \Delta)}{c : (\Gamma_1, x_2^\varepsilon : \varepsilon, x_1^\varepsilon : \varepsilon, \Gamma_2 \vdash \Delta)} \text{(pc\vdash)}
 \end{array}$$

Figure .4.1.c: Structural rules (expressions)

$$\begin{array}{c}
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta}{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \alpha^\varepsilon : \varepsilon, \Delta} \text{(\vdash wt)} \qquad \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \alpha_1^\varepsilon : \varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta}{\Gamma \vdash \underline{t_{\varepsilon_0} [\beta^\varepsilon / \alpha_1^\varepsilon, \beta^\varepsilon / \alpha_2^\varepsilon] : \varepsilon_0} \mid \beta^\varepsilon : \varepsilon, \Delta} \text{(\vdash ct)} \\
 \\
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta}{\Gamma, x^\varepsilon : \varepsilon \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta} \text{(wt\vdash)} \qquad \frac{\Gamma, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta}{\Gamma, x^\varepsilon : \varepsilon \vdash \underline{t_{\varepsilon_0} [x^\varepsilon / x_1^\varepsilon, x^\varepsilon / x_2^\varepsilon] : \varepsilon_0} \mid \Delta} \text{(ct\vdash)} \\
 \\
 \frac{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta_1, \alpha_1^\varepsilon : \varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta_2}{\Gamma \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta_1, \alpha_2^\varepsilon : \varepsilon, \alpha_1^\varepsilon : \varepsilon, \Delta_2} \text{(\vdash pt)} \qquad \frac{\Gamma_1, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon, \Gamma_2 \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta}{\Gamma_1, x_2^\varepsilon : \varepsilon, x_1^\varepsilon : \varepsilon, \Gamma_2 \vdash \underline{t_{\varepsilon_0} : \varepsilon_0} \mid \Delta} \text{(pt\vdash)}
 \end{array}$$

Figure .4.1.d: Structural rules (evaluation contexts)

$$\begin{array}{c}
\frac{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta}{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \alpha^\varepsilon : \varepsilon, \Delta} \text{ (}\vdash\text{we)} \qquad \frac{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \alpha_1^\varepsilon : \varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta}{\Gamma \mid \underline{e_{\varepsilon_0} [\beta^\varepsilon / \alpha_1^\varepsilon, \beta^\varepsilon / \alpha_2^\varepsilon] : \varepsilon_0} \vdash \beta^\varepsilon : \varepsilon, \Delta} \text{ (}\vdash\text{ce)} \\
\\
\frac{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta}{\Gamma, x^\varepsilon : \varepsilon \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta} \text{ (we}\vdash\text{)} \qquad \frac{\Gamma, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta}{\Gamma, x^\varepsilon : \varepsilon \mid \underline{e_{\varepsilon_0} [x^\varepsilon / x_1^\varepsilon, x^\varepsilon / x_2^\varepsilon] : \varepsilon_0} \vdash \Delta} \text{ (ce}\vdash\text{)} \\
\\
\frac{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta_1, \alpha_1^\varepsilon : \varepsilon, \alpha_2^\varepsilon : \varepsilon, \Delta_2}{\Gamma \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta_1, \alpha_2^\varepsilon : \varepsilon, \alpha_1^\varepsilon : \varepsilon, \Delta_2} \text{ (}\vdash\text{pe)} \qquad \frac{\Gamma_1, x_1^\varepsilon : \varepsilon, x_2^\varepsilon : \varepsilon, \Gamma_2 \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta}{\Gamma_1, x_2^\varepsilon : \varepsilon, x_1^\varepsilon : \varepsilon, \Gamma_2 \mid \underline{e_{\varepsilon_0} : \varepsilon_0} \vdash \Delta} \text{ (pe}\vdash\text{)}
\end{array}$$

Figure .4.1.e: General shape of logic rules

$$\begin{array}{c}
\frac{\Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : \varepsilon_1} \mid \Delta_1 \quad \dots \quad \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : \varepsilon_q} \mid \Delta_q}{\Gamma_{q+1} \mid \underline{s_{\varepsilon_{q+1}}^1 : \varepsilon_{q+1}} \vdash \Delta_{q+1} \quad \dots \quad \Gamma_{q+r} \mid \underline{s_{\varepsilon_{q+r}}^r : \varepsilon_{q+r}} \vdash \Delta_{q+r}} \left(\mathfrak{s}_k^{\tau_-^j} \vdash \right) \\
\frac{\Gamma_1, \dots, \Gamma_{q+r} \mid \underline{\mathfrak{s}_k^{\tau_-^j}(v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1, \dots, s_{\varepsilon_{q+r}}^r) : -} \vdash \Delta_1, \dots, \Delta_{q+r}}{\Gamma \vdash \underline{\mu \langle \mathfrak{s}_1^{\tau_-^j}(\vec{x}_1, \vec{\alpha}_1).c_1 \mid \dots \mid \mathfrak{s}_l^{\tau_-^j}(\vec{x}_l, \vec{\alpha}_l).c_l \rangle : -} \mid \Delta} \text{ (}\vdash\tau_-^j\text{)} \\
\\
\frac{\Gamma_1 \vdash \underline{v_{\varepsilon_1}^1 : \varepsilon_1} \mid \Delta_1 \quad \dots \quad \Gamma_q \vdash \underline{v_{\varepsilon_q}^q : \varepsilon_q} \mid \Delta_q}{\Gamma_{q+1} \mid \underline{s_{\varepsilon_{q+1}}^1 : \varepsilon_{q+1}} \vdash \Delta_{q+1} \quad \dots \quad \Gamma_{q+r} \mid \underline{s_{\varepsilon_{q+r}}^r : \varepsilon_{q+r}} \vdash \Delta_{q+r}} \left(\vdash \mathfrak{v}_k^{\tau_+^j} \right) \\
\frac{\Gamma_1, \dots, \Gamma_q \vdash \underline{\mathfrak{v}_k^{\tau_+^j}(v_{\varepsilon_1}^1, \dots, v_{\varepsilon_q}^q, s_{\varepsilon_{q+1}}^1, \dots, s_{\varepsilon_{q+r}}^r) : +} \mid \Delta_1, \dots, \Delta_q}{c_1 : (\Gamma, \vec{x}_1 : \vec{\varepsilon}_1 \vdash \vec{\alpha}_1 : \vec{\varepsilon}_1, \Delta) \quad \dots \quad c_l : (\Gamma, \vec{x}_l : \vec{\varepsilon}_l \vdash \vec{\alpha}_l : \vec{\varepsilon}_l, \Delta)} \left(\tau_+^j \vdash \right) \\
\frac{\quad}{\Gamma \mid \underline{\tilde{\mu} [\mathfrak{v}_1^{\tau_+^j}(\vec{x}_1, \vec{\alpha}_1).c_1 \mid \dots \mid \mathfrak{v}_l^{\tau_+^j}(\vec{x}_l, \vec{\alpha}_l).c_l] : +} \vdash \Delta}
\end{array}$$

Figure .4.1.f: Logic rules for multiplicative types

$$\begin{array}{c}
\frac{c:(\Gamma, x^+ : + \vdash \alpha^- : -, \Delta)}{\Gamma \vdash \underline{\mu(x^+ \cdot \alpha^-).c} : - \mid \Delta} (\rightarrow) \quad \frac{\Gamma_1 \vdash \underline{v_+ : +} \mid \Delta_1 \quad \Gamma_2 \mid \underline{s_- : -} \vdash \Delta_2}{\Gamma_1, \Gamma_2 \mid \underline{v_+ \cdot s_- : -} \vdash \Delta_1, \Delta_2} (\rightarrow\vdash) \\
\\
\frac{c:(\Gamma \vdash \alpha^- : -, \beta^- : -, \Delta)}{\Gamma \vdash \underline{\mu(\alpha^- \wp \beta^-).c} : - \mid \Delta} (\wp) \quad \frac{\Gamma_1 \mid \underline{s_-^1 : -} \vdash \Delta_1 \quad \Gamma_2 \mid \underline{s_-^2 : -} \vdash \Delta_2}{\Gamma_1, \Gamma_2 \mid \underline{(s_-^1 \wp s_-^2) : -} \vdash \Delta_1, \Delta_2} (\wp\vdash) \\
\\
\frac{\Gamma_1 \vdash \underline{v_+^1 : +} \mid \Delta_1 \quad \Gamma_2 \vdash \underline{v_+^2 : +} \mid \Delta_2}{\Gamma_1, \Gamma_2 \vdash \underline{(v_+^1 \otimes v_+^2) : +} \mid \Delta_1, \Delta_2} (\otimes) \quad \frac{c:(\Gamma, x^+ : +, y^+ : + \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}(x^+ \otimes y^-).c} : + \vdash \Delta} (\otimes\vdash) \\
\\
\frac{c:(\Gamma \vdash \Delta)}{\Gamma \vdash \underline{\mu\tilde{()}.c} : - \mid \Delta} (\perp) \quad \frac{}{\mid \underline{\tilde{()}} : - \vdash} (\perp\vdash) \\
\\
\frac{}{\vdash \underline{()}: + \mid} (1\vdash) \quad \frac{c:(\Gamma \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}().c} : + \vdash \Delta} (1)
\end{array}$$

Figure .4.1.g: Logic rules for additive types

$$\begin{array}{c}
\frac{c_1:(\Gamma \vdash \alpha_1^- : -, \Delta) \quad c_2:(\Gamma \vdash \alpha_2^- : -, \Delta)}{\Gamma \vdash \underline{\mu\langle(\pi_1 \cdot \alpha_1^-).c_1 \mid (\pi_2 \cdot \alpha_2^-).c_2\rangle} : - \mid \Delta} (\&) \quad \frac{\Gamma \mid \underline{s_- : -} \vdash \Delta}{\Gamma \mid \underline{\pi_i \cdot s_- : -} \vdash \Delta} (\&\vdash) \\
\\
\frac{\Gamma \vdash \underline{v_+ : +} \mid \Delta}{\Gamma \vdash \underline{l_i(v_+): +} \mid \Delta} (\oplus) \quad \frac{c_1:(\Gamma, x_1^+ : + \vdash \Delta) \quad c_2:(\Gamma, x_2^+ : + \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}[l_1(x_1^+).c_1 \mid l_2(x_2^+).c_2] : +} \vdash \Delta} (\oplus\vdash) \\
\\
\frac{}{\Gamma \vdash \underline{\mu\langle\rangle} : - \mid \Delta} (\top) \quad (\text{No } (\top\vdash) \text{ rule}) \\
\\
(\text{No } (\vdash 0) \text{ rule}) \quad \frac{}{\Gamma \mid \underline{\tilde{\mu}[]} : + \vdash \Delta} (0\vdash)
\end{array}$$

Figure .4.1.h: Logic rules for shifts

$$\begin{array}{cc}
 \frac{c:(\Gamma \vdash \alpha^+ : +, \Delta)}{\Gamma \vdash \underline{\mu\{\alpha^+\}.c : -} \mid \Delta} (\vdash\Uparrow) & \frac{\Gamma \mid \underline{s_+ : +} \vdash \Delta}{\Gamma \mid \underline{\{s_+\} : -} \vdash \Delta} (\Uparrow\vdash) \\
 \\
 \frac{\Gamma \vdash \underline{v_- : -} \mid \Delta}{\Gamma \vdash \underline{\{v_-\} : +} \mid \Delta} (\vdash\Downarrow) & \frac{c:(\Gamma, x^- : - \vdash \Delta)}{\Gamma \mid \underline{\tilde{\mu}\{x^-\}.c : +} \vdash \Delta} (\Downarrow\vdash)
 \end{array}$$

Figure .4.1.i: Logic rules for negations

$$\begin{array}{cc}
 \frac{c:(\Gamma, x^+ : + \vdash \Delta)}{\Gamma \vdash \underline{\mu_{\neg-}(x^+).xc : -} \mid \Delta} (\vdash\neg_-) & \frac{\Gamma \vdash \underline{v_+ : +} \mid \Delta}{\Gamma \mid \underline{\neg_-(v_+) : -} \vdash \Delta} (\neg_-\vdash) \\
 \\
 \frac{\Gamma \mid \underline{s_- : -} \vdash \Delta}{\Gamma \vdash \underline{\neg_+(s_-) : +} \mid \Delta} (\vdash\neg_+) & \frac{c:(\Gamma \vdash \alpha^- : -, \Delta)}{\Gamma \mid \underline{\tilde{\mu}_{\neg+}(\alpha^-).ac : +} \vdash \Delta} (\neg_+\vdash)
 \end{array}$$




Contents

0. Introduction	2
0.1. Motivation	3
0.2. Background	5
0.2.1. Calculi	5
λ -calculi and Call-by-push-value	5
The $\bar{\lambda}\mu\tilde{\mu}$ -calculus	5
Polarized sequent calculi	6
0.2.2. Solvability in arbitrary programming languages	6
Observational equivalence and preorder	6
Operational relevance and solvability	6
The central role of unsolvability	7
Unary operational completeness	7
Operational characterization of solvability	8
0.2.3. Solvability in λ -calculi	8
Call-by-name solvability	9
Call-by-value solvability	10
0.3. Content	11
0.4. Notations	12
Reduction sequences	12
Main reductions	12
Closure of reductions under contexts	12
0.5. Table of contents	14
 A. Introduction to L calculi	 15
Content	16
Contribution	16
 I. Pure call-by-name calculi	 18
Summary	18
Table of contents	19
I.1. A pure call-by-name λ -calculus: $\lambda_{\mathbf{N}}^{\rightarrow}$	20
Syntax	20
Contexts	20
Substitutions and disubstitutions	21
β -reduction	22
σ -reductions	23

Contents

	η -expansion	25
I.2.	A pure call-by-name λ -calculus with toplevel focus: $\underline{\lambda}_N^{\rightarrow}$	27
	Searching for the next redex	27
	Simulation	28
	Refocusing	28
	Properties of reductions	29
I.3.	A pure call-by-name abstract machine: M_N^{\rightarrow}	30
	The inside-out syntax	30
	Disubstitutions	30
	Ambiguity of the ambient calculus	33
I.4.	Equivalence between $\underline{\lambda}_N^{\rightarrow}$ and M_N^{\rightarrow}	35
	Inside-out and outside-out descriptions	35
	Translations	36
	Substitutions	40
	Disubstitutions	42
	Reductions	43
I.5.	Translations between λ_N^{\rightarrow} and $\underline{\lambda}_N^{\rightarrow}$	46
	Focus insertion and erasure	46
	Reductions through focus erasure	47
	Reductions through focus insertion	48
I.6.	A pure call-by-name λ -calculus with focus: $\underline{\lambda}_n^{\rightarrow}$	51
I.6.1.	The simple fragment of the naive $\underline{\lambda}_n^{\rightarrow}$ calculus	51
	Decomposing the strong reduction	51
	Focus erasure in place of focus movement	53
I.6.2.	The naive $\underline{\lambda}_n^{\rightarrow}$ calculus	53
	Stack deferrals	53
	\triangleright_{μ} as a generalization of \triangleright_m and \boxtimes	54
	Underlines as potential places of interaction	55
	Reducing let-expressions	56
	Undesirable strong reductions	56
I.6.3.	The $\underline{\lambda}_n^{\rightarrow}$ calculus	57
	Explicit command boundaries	57
	Coercions	59
	Evaluation contexts	59
	Disubstitution	59
	Reductions	60
I.7.	Translations between λ_N^{\rightarrow} and $\underline{\lambda}_n^{\rightarrow}$	61
I.8.	A pure call-by-name intuitionistic L calculus: Li_n^{\rightarrow}	62
I.8.1.	From the M_N^{\rightarrow} abstract machine to the Li_n^{\rightarrow} calculus	62
	Decomposing the strong reduction	62
	Pattern matching stacks	62
	Stack variable names	62
	Binding the stack variable	63


Contents

I.8.2.	The $\text{Li}_n^{\rightarrow}$ calculus	65
	Let-expressions and $\tilde{\mu}$	65
	Coercions	65
	Disubstitutions	65
	Reductions	65
I.9.	Equivalence between λ_n^{\rightarrow} and $\text{Li}_n^{\rightarrow}$	67
I.10.	A pure call-by-name classical L calculus: L_n^{\rightarrow}	68
I.11.	Simply-typed L calculi	69
II.	Pure call-by-value calculi	70
II.1.	A pure call-by-value λ -calculus: λ_V^{\rightarrow}	71
II.2.	A pure call-by-value λ -calculus with focus: λ_V^{\rightarrow}	72
II.3.	A pure call-by-value intuitionistic L calculus: $\text{Li}_V^{\rightarrow}$	73
II.4.	A pure call-by-value classical L calculus: L_V^{\rightarrow}	74
B.	Untyped polarized calculi	75
	Introduction 	76
	Content 	77
	Contribution 	77
III.	Pure polarized calculi	78
III.1.	Relative expressiveness of call-by-name and call-by-value	79
III.1.1.	Embedding call-by-name in call-by-value	79
	The $\lambda_V^{\rightarrow\uparrow}$ -calculus	79
	Embedding $\lambda_V^{\rightarrow\uparrow}$ in λ_V^{\rightarrow}	79
	Embedding λ_N^{\rightarrow} in $\lambda_V^{\rightarrow\uparrow}$	81
III.1.2.	Embedding call-by-value in call-by-name	82
	The $\lambda_N^{\rightarrow\downarrow}$ -calculus	82
	Embedding $\lambda_N^{\rightarrow\downarrow}$ in $\lambda_N^{\rightarrow\otimes}$	82
	Embedding λ_V^{\rightarrow} in $\lambda_N^{\rightarrow\downarrow}$	84
III.2.	A pure polarized λ -calculus: $\lambda_P^{\rightarrow\uparrow\downarrow}$	85
III.3.	A pure polarized λ -calculus with focus: $\lambda_P^{\rightarrow\uparrow\downarrow}$	86
III.4.	A pure polarized intuitionistic L-calculus: $\text{Li}_P^{\rightarrow\uparrow\downarrow}$	87
III.5.	A pure polarized classical L-calculus: $\text{L}_P^{\rightarrow\uparrow\downarrow}$	88
IV.	Polarized calculi with pairs and sums	89
IV.1.	A polarized λ -calculus with pairs and sums: $\lambda_P^{\rightarrow\&\uparrow\otimes\oplus\downarrow}$	90
IV.2.	CBPV as a subcalculus of $\lambda_P^{\rightarrow\&\uparrow\otimes\oplus\downarrow}$	91
IV.2.1.	CBPV	91
	Syntax	91
	Operational semantics	91
	Complex values	91

Contents

IV.2.2. Embedding CBPV into $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$	95
Embedding values and computations	95
Differences between CBPV and $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$	95
Complex values and positive expressions	97
Preservation of operational semantics	97
IV.3. A polarized λ -calculus with focus: $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$	99
IV.4. A polarized intuitionistic L calculus: $Li_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$	100
IV.5. A polarized classical L calculus: $L_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$	101
IV.6. The CBPV abstract machine as a subcalculus of $\lambda_p^{\rightarrow \& \uparrow \otimes \oplus \Downarrow}$	102
V. Polarized calculi with arbitrary constructors	103
V.1. A (classical) polarized L-calculus: $L_p^{\bar{\tau}}$	104
V.1.1. Syntax	104
Type formers	104
Value and stack constructors	104
Syntax	107
V.1.2. Reductions	107
Definitions	107
Normal forms, clashes and waiting commands	113
Properties	116
V.1.3. Well-typed and well-polarized terms	118
Well-typed terms	118
Alternative presentations	123
Well-polarized terms	124
V.2. Intuitionistic and minimalistic polarized L-calculi: $Li_p^{\bar{\tau}}$ and $Lm_p^{\bar{\tau}}$	126
V.2.1. Intuitionistic and minimalistic fragments	126
Fragment definitions	126
Inoperable rules	131
Inclusions	132
Straightforwardly minimalistic type formers	134
V.2.2. A syntax for the minimalistic fragment	136
Characterization of $Lm_p^{\bar{\tau}}$ via free stack variables	136
Output polarities	136
A BNF grammar for $Lm_p^{\bar{\tau}}$	137
V.2.3. Properties	144
Disubstitutions	144
Reductions	145
V.3. A polarized λ -calculus with focus equivalent to $Lm_p^{\bar{\tau}}$: $\lambda_p^{\bar{\tau}}$	147
V.4. Equivalence between $\lambda_p^{\bar{\tau}}$ and $Lm_p^{\bar{\tau}}$	148
V.5. A polarized λ -calculus: $\lambda_p^{\bar{\tau}}$	149
VI. Dynamically typed polarized calculi	150
VI.1. Clashes and dynamically typed calculi	151

Contents

VI.2. A dynamically typed polarized λ -calculus: $\lambda_p^{\mathcal{PN}}$	152
VI.3. A dynamically typed polarized λ -calculus with focus: $\lambda_p^{\mathcal{PN}}$	153
VI.4. A dynamically typed polarized intuitionistic L calculus: $\text{Li}_p^{\mathcal{PN}}$	154
VI.5. A dynamically typed polarized classical L calculus: $L_p^{\mathcal{PN}}$	155
C. Solvability in polarized calculi	156
Content	157
Contribution	157
Introduction to solvability and operational completeness 	158
Goal and content	158
Caveats	158
Summary	158
C.1. A meaning for programs	158
Programs as maps from inputs to outputs	158
Comparing programs	161
C.2. A compositional meaning for fragments	162
Fragments and plugging	162
Program-preserving observational preorder and equivalence	165
Program preservation: a displeasingly strong syntactic invariant	170
Weakening syntactic invariants	171
C.3. Distinguishability, separability and binary operational completeness	173
C.4. Operational relevance, solvability and unary operational completeness	178
VII.Call-by-name solvability	183
Three notions of observational equivalence induced by three notions of evaluation	184
Two notions of operational relevance and one notion of solvability	185
Convergence testing as a means to bootstrap meaningful definitions	186
The usefulness of L-calculi	187
VII.1.Reductions and induced notions of evaluation	188
Five reductions	188
Normal forms	192
Substitutivity and disubstitutivity	193
Determinism, confluence, and uniqueness of termination behavior	195
Three notions of evaluation	196
Existence of normal forms and evaluation	198
VII.2.Usefulness of the trivial interpretation of programs	202

Contents

VII.3. Instanciating the general definitions	203
VII.3.1. Parameters	203
Terms and states	203
Closedness	203
Parameters	204
VII.3.2. A meaning for programs	205
Main definition	205
Alternative definitions	205
Properties	206
VII.3.3. Observational preorder and equivalence	207
Main definitions	207
Alternative definitions	209
Properties	211
Transitivity	211
Monotonicity	212
Soundness of β -conversion	213
VII.3.4. Operational relevance and solvability	214
VII.4. Equivalences between definitions	215
VIII. Call-by-value solvability	216
IX. Polarized solvability	217
Bibliography	218
.1. Properties of disubstitutions	225
.2. Properties of reductions	228
.3. Detailed proofs	229
.4. Extra figures	234