On the lambda calculus with constructors

Barbara Petit

N° d'ordre: 628
N° attribué par la bibliothèque: __ENSL628

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon — École Normale Supérieure de Lyon

spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme

École doctorale de mathématiques et informatique fondamentale

présentée et soutenue publiquement le 13/07/11

par Mademoiselle Barbara PETIT

Titre :

Autour du lambda calcul avec constructeurs.

Directeur de thèse : Monsieur Alexandre MIQUEL

Après avis de : *Monsieur Antonio BUCCIARELLI*
Monsieur Luke ONG

Devant la commission d'examen formée de :
Monsieur Antonio BUCCIARELLI, Membre/Rapporteur
Monsieur Horatiu CIRSTEA, Membre
Monsieur Guillaume HANROT, Membre
Monsieur Hugo HERBELIN, Membre
Monsieur Alexandre MIQUEL, Membre
Monsieur Luke ONG, Membre/Rapporteur

A mes Parents,

parce que c'est les plus forts du monde...

Merci qui?

Alexandre, tout d'abord,
pour son implication dans l'encadrement de cette thèse, pour son recul scientifique sur le sujet et ses qualités pédagogiques.

Plume, le LIP, et puis aussi PPS,
qui m'ont offert un cadre de travail idéal: des chercheurs attentifs au travail des doctorants, une activité scientifique dynamique, le tout dans une ambiance conviviale, favorisée par l'efficacité (et la patience!) des secrétaires et des admin-sys.

Les Lamas, la Choco'team, le LIG,
tous ceux avec qui j'ai eu l'occasion de travailler, et notamment:

Alejandro, pour m'avoir intéressée aux calculs quantiques.

Paolo, pour l'expertise Tikz et les dessins de catégories, entre autre.

Colin, pour les tordages de cerveau devant le tableau barbouillé, qui précèdent l'arrivée de la lumière.

Olivier, pour les cours de logique linéaire et sa disponibilité pour des discutions au tableau avec les doctorants.

Les proggirls, Christine et Séverine, pour les séances devant ordi et surtout pour les autres.

Elsa,
pour l'accueil enthousiaste lors des mes "mission sans frais d'hébergement" à Paris. Faten et Mehdi pour les chouettes soirées à ces occasions.

Nache et Jano,
pour les mates et tout le reste, lors de mes missions à Grenoble (j'espère qu'il y en aura d'autres).

Cliss, et la bibliothèque de Gennevilliers, pour leur bonne ambiance de rédaction de thèse.

Enfin et surtout, puisque la thèse c'est toujours avec le moral en orange et violet:

Tous ceux que j'aime, et qui font que la vie elle chante, à pleins poumons. Aunque estén lejos, me pone pila en la vida conocerlos!

Contents

Chapter 1

Introduction

The lambda calculus was introduced in the early 30s, by Alonzo Church and his student Stephen Kleene. At this time several mathematicians were interested in a question raised by Hilbert in 1928: the *Entscheidungsproblem* (*decision problem*, in German). At a time when computers were only prototypes of analog computers or punch cards engines, Hilbert was putting out the challenge of finding a way to solve *automatically* any problem (expressed in a formal language). He had the dream of a universal algorithm that would respond "true"or "false" (in a reliable way, but not necessarily justified by a proof) to any mathematical question, even those that mathematicians were not able to solve.

Many logicians were thus trying to establish a formal framework to define automatic computing. Three different formalisms emerged almost simultaneously: the mu-recursive functions (developed by Herbrand and Gödel [Gö34] and then reformulated by Kleene [Kle36a]), Church's lambda calculus [Chu32] and Turing machines [Tur36]. These three notions of computability were found to be equivalent [Kle36b, Tur37]. Church's thesis was then fully meaningful: the problems that have automatically computable solutions exactly correspond to the mathematical functions expressible in one of these three formal systems (or an equivalent system, that would be called *Turing-complete* nowadays). The *Entscheidungsproblem* was thereby negatively answered [God33, Chu36, Tur36]: no algorithm can solve *any* arithmetic question (and there is *a fortiori* no universal algorithm).

Lambda calculus. Among these Turing-complete systems (whose computational power is still not exceeded by modern computers), the lambda calculus is remarkable for its simplicity. It relies on a central idea:

Everything is a function.

As simple to express as it may be, this idea is far from intuitive. Take for example an arithmetic expression $2 \times (3+4x)$. One can abstract this expression *w.r.t.* the variable x, and obtain the function that maps the formal argument x to the value of $2 \times (3+4x)$. In lambda calculus, this function is denoted by $\lambda x.2 \times (3+4x)$.

To compute its value at 7 for example, one applies the real argument 7 to the function. The first step of the computation consists in replacing the formal argument x with the real argument 7:

$$(\lambda x.2 \times (3 + 4x))\ (7) \quad \rightsquigarrow \quad 2 \times (3 + 4 \times 7)\ .$$

This simple step is called the "beta"-reduction. Then, the "real" computation is performed:

$$2 \times (3 + 4 \times 7) \quad \rightsquigarrow \quad 62\ .$$

With the lambda calculus, Church's *tour de force* consists in decomposing each step of pure computation into β-reductions, through some encodings. There are no more integers nor arithmetic operations, but simply *variables*, *abstractions* of an expression relatively to a variable, and *applications* of a function to an argument. Computing then amounts to replacing some variables by some expressions (by *terms*).

Terms (written t, u *etc.*) can be constructed either with variables (denoted by x, y, z), or by abstracting a term t *w.r.t.* a variable x, or by applying a term t to another term u. In the so-called BNF notation, this is expressed as

$$t, u \ ::= \quad x \ \mid \ \lambda x.t \ \mid \ tu\ .$$

Just like an expression $2\times(3+4x)$ may have an unknown x, a term may have a *free* variable. On the contrary, x is not an unknown in the function $\lambda x.2\times(3+4x)$, but a formal argument. We say that x is *bounded* in $\lambda x.2 \times (3 + 4x)$ (that is why the λ is sometimes called a variable *binder*). The substitution of the variable x by the argument a in the expression e, written $e\ [x := a]$, obviously replaces only the free occurrences of x in e by a. When a function $\lambda x.t$ is applied to an argument u, this substitution is performed by the β-reduction:

$$(\lambda x.t)\ u \quad \rightarrow \quad t[x := u] \qquad (\beta)$$

Since the function mapping x to $2\times(3+4x)$ is the same than the one mapping y to $2 \times (3 + 4y)$, we can freely rename the bounded variables of a term:

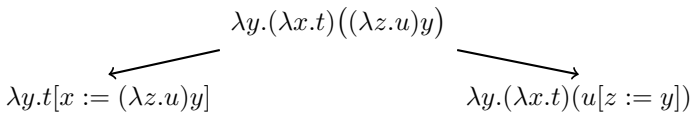
$$\lambda x.t \ \simeq_\alpha \ \lambda y.(t[x := y])\ .$$

In the lambda calculus this is called α-*conversion*. Terms are generally considered up to α-equivalence.

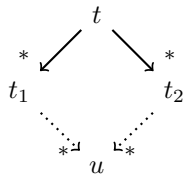
The lambda calculus, like any other formal language, raises several natural questions:

Confluence. A term is said to be confluent if any computation starting from it leads to the same result. In the expression $2 \times (3 + 4 \times 7)$ for example, we can simplify the multiplication 4×7 and get $2 \times (3 + 28)$, or we can distribute the multiplication by 2 over the addition and get $2\times3+2\times4\times7$.

The arithmetic is confluent: we know that if we continue the computation of each expression, we will obtain the same result (62). Thus the choice of the computational strategy does not matter. In the lambda calculus, a term can also have several potential reductions, as this example shows:



It appears that the lambda calculus is confluent too [Bar84]: whatever reduction we choose for a given term, we will always be able to continue the computation until a common reduct:



Termination. When performing a computation on an expression, it might be interesting to know whether it will terminate (and return a value) or not. More precisely, given a term t, we may wonder if:

- there is a reduction of t that leads to a value (or to a non-reducible term). In this case we say that t is *normalising*. In arithmetic, for example, the expression $2 \times (3 + 4)$ is normalising:

$$2 \times (3 + 4) \rightsquigarrow 2 \times 7 \rightsquigarrow 14$$

- every reduction of t necessarily stops. We then say that t is *strongly normalising*. If we consider the rules of distributivity of the multiplication over the addition and the factorisation, then the expression $2 \times (3 + 4)$ is not strongly normalising:

$$2 \times (3+4) \rightsquigarrow 2 \times 3 + 2 \times 4 \rightsquigarrow 2 \times (3+4) \rightsquigarrow 2 \times 3 + 2 \times 4 \rightsquigarrow \cdots$$

The confluence and the strong normalisation of a system are interesting because the reduction strategy is irrelevant in such systems. No matter which way we apply the reduction rules, the computation always yields a value, which is always the same one.

In the lambda calculus there are some non normalising terms (that *a fortiori* are not strongly normalising). The most notorious one is

$$\Omega \;=\; (\lambda x.xx)\ (\lambda x.xx)\ .$$

It necessarily reduces to itself. However, it is possible to restrict the set of terms that we consider with a *typing system*, that selects only strongly normalising terms.

Types. In some programming languages, the programs are annotated (by a type), so as to guide the programmer. For instance, if a program computes the successor of an integer given as a parameter, the Boolean "true" has to be prohibited as input. Similarly the logical negation function (which returns "false" on the input "true", and vice versa), cannot be applied to the integer 3. To deal with this, some basic types are defined (such as *nat* for the integers, or *bool* for the Booleans). It is then set out, in the annotation of the successor function, that is only accepts arguments of type *nat*.

A function expecting to receive an input of type A and to return a result of type B will be assigned the type $A \to B$. In simply typed lambda calculus, a type is defined by the following syntax:

$$A, B \ ::= \quad \alpha \ \mid \ A \to B$$

(where α is a basic type). Thus if *nat* is a type for the integers, the successor function has type $nat \to nat$:

$$Succ : \ nat \to nat \ .$$

In order to apply an argument of type A to a function, it must be ensured that it is annotated with a type $A \to B$. This produces a result of type B. In the simply typed lambda calculus, we thus have the following typing rule:

$$\frac{t : A \to B \quad u : A}{tu : B}$$

(above the horizontal bar are the assumptions, or *premises* of the typing rule, and below it is the conclusion).

Notice that it is not always possible to determine the type of a term *in absolute*. Assigning a type to $Succ\,(x)$ for instance, requires x to be of type *nat*. Yet a free variable has no type *a priori*. Therefore some assumptions are necessary concerning the type of the free variables of a term. We use a *typing* context (usually written as Γ) that to some variables x_i assigns a corresponding type A_i:

$$\Gamma \ = \quad x_1 : A_1, \ldots, x_n : A_n \ .$$

If a term t has some free variables $x_1, \ldots, x_n$, then we write

$$\Gamma \vdash t : B$$

for "u has type B under the assumption Γ"(*i.e.* under the assumption that each variable x_i is of type A_i).

If a term t has type B under the assumption that x is of type A, then it can be abstracted *w.r.t.* the variable x. This provides a function taking arguments of type A and returning a result of type B:

$$\frac{x : A \vdash t : B}{\lambda x.t : A \to B}$$

Also the simply typed lambda calculus consists in typing the lambda terms using the following typing rules:

$$\frac{x_i : A_i \ \in \Gamma}{\Gamma \vdash x_i : A_i} \quad ; \quad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \quad ; \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

Not all lambda terms are typable. In particular, there is no type for the diverging term $\Omega = (\lambda x.xx)\ (\lambda x.xx)$. Actually this type system only types strongly normalising terms.

Encoding. The expressiveness of a Turing-complete language is achieved in the lambda calculus through some encodings. An integer n, for instance, is represented by a term taking as input a function f and an argument x, and applying n times the function f at x:

$$\begin{array}{rcl} 0 & = & \lambda f.\lambda x.x \\ 1 & = & \lambda f.\lambda x.fx \\ 2 & = & \lambda f.\lambda x.f(fx) \\ 3 & = & \lambda f.\lambda x.f(f(fx)) \\ & \vdots & \end{array}$$

The successor function is then written like a function mapping an integer n, to the integer $n+1$, that is: the function that takes as input a function f and an argument x, and that applies f to x *one more time* than n:

$$\mathit{Succ} \ = \ \lambda n.\lambda f'.\lambda x'.f'(n\ f'\ x')\ .$$

We can check that β-reduction actually computes the successor of 1:

$$\begin{array}{rcl} \mathit{Succ}\ 1 & = & \big(\lambda n.\lambda f'.\lambda x'.f'(n\ f'\ x')\big)\ \big(\lambda f.\lambda x.fx\big) \\ & \to & \lambda f'.\lambda x'.f'(n\ f'\ x')[n := \lambda f.\lambda x.fx] \\ & = & \lambda f'.\lambda x'.f'\big((\lambda f.\lambda x.fx)\ f'\ x'\big) \\ & \to & \lambda f'.\lambda x'.f'\big((fx)\ [f := f'][x := x']\big) \\ & = & \lambda f'.\lambda x'.f'\big(f'x'\big) \end{array}$$

Up to α-equivalence, $\mathit{Succ}\ 1$ becomes $\lambda f.\lambda x.f(fx)$, *i.e.* 2.

In the same way, Booleans, pairs *etc.* can be encoded in the pure lambda calculus. However, such encodings soon become tiresome, especially when computing more elaborate functions than the successor of an integer. That is why the so-called *functional* programming languages (*i.e.* with no side effect) that are based on the lambda calculus (such as LISP [BB64] or ML [MTH90]) are provided with some primitive constructions for the usual data structures, like Booleans and integers. Hence it makes sense to add some constructors $\mathsf{0}$ and S to the syntax of the lambda calculus, and then to represent unary integers by

$$n \ := \quad \mathsf{0} \ \mid \ \mathsf{S}(n)\ .$$

Then comes the question of the interaction of such data structures with the program environment (the *context*). Just like a function can generate a computation when it is *applied* to an argument, a data structure can be *analysed* by a *pattern matching* construct. Depending on the programming language, the pattern matching mechanism can be more or less complex. The `case` instruction in Pascal for instance, is rather basic. It can only match constants:

```
case i of
  0 :  First_result ;
  1 :  Second_result ;
  ...
end;
```

On the other hand, the pattern matching *à la* ML can capture the arguments of a constructor, and us them in the output. For example the predecessor function is defined that way:

```
fun pred (n) =
  case n of
    ZERO   => ZERO
  | SUCC p => p
```

Pattern matching. Whereas the first functional programming languages have emerged from the late 50's [McC60], pattern matching has been hardly studied in the theoretical framework of lambda calculus before the 90's [vO90]. Several approaches were then developed, among which:

- The rewriting calculus, or ρ-calculus [CK98, Cir01] was introduced in the late 90s. In addition to lambda terms it has data structures, that are built with constants. Like any term, these constants are patterns, and so they can be analysed by a pattern matching construct. Their free variables can then be instantiated, and so any rewriting rule can be simulated. The calculus is confluent if the set of terms is restricted to the ones that enjoy the *rigid pattern condition*. (a notion that was introduced earlier by Van Oostrom [vO90]).

  There is a typed version of the ρ-calculus [CK00], in which pattern matching is indexed by the typing context. The type system ensures the strong normalisation property and the subject reduction.

- The *Pattern calculus* appeared in the early 00's [Jay04], together with a polymorphic type system. A matchable pattern is either a variable (which can be instantiated by any term —in that case pattern matching is in fact the usual β-reduction) or a constant, or an application. This calculus is confluent (subject to first reduce a term in head normal form before matching it) and enjoys subject reduction. However it does not avoid match failure, and the reduction rules are restricted to typed terms. It should be noted

that type inference is decidable for this system, and its implementation gave birth to the programming language Bondi [Jay09].

There is also an untyped version of this calculus [JK06]. The reduction is there parametrised by a set of free variables (that can be seen as constructors), and prevents match failure during the evaluation of closed terms (they are intuitively caught by the identity function). What is more, this calculus is confluent too.

In the pattern calculus, as well as in ρ-calculus, pattern matching uses a "generalised" substitution operation. Thus a unification algorithm is necessary for matching a term with a pattern during the evaluation.

- The (untyped) lambda calculus with constructors appeared in the mid-00's [AMR06]. Pattern matching is only performed on constants, and the expressiveness of ML pattern matching is reached thanks to some commutation rules. Unlike the two previous calculi, pattern matching does not use any meta-operation. In addition it can deal with variadic constructors (avoiding the problem of partial application for data structures). The calculus was proved confluent, and moreover it satisfies the *separation* property (in the spirit of Böhm's theorem [BDCPR79]).

  A polymorphic type system (that we present in Chap. 2.2) was proposed for this calculus [Pet11]. It ensures strong normalisation and the absence of match failure during execution for a restriction of the calculus (the restriction consists in removing an "unessential" reduction rule).

None of these calculi can match non-linear patterns (such as patterns of the form $\mathtt{c}xx$, checking that the two arguments of `c` are identical), or to define the "parallel"-or (also called `por` [Plo77], which returns true if at least one of its arguments is true, even if the evaluation of the other one does not terminate).

Semantics. As far as we know, these calculus with pattern matching features had never been studied from the point of view of the denotational semantics. Defining a denotational model for a language consists in embedding its syntax into some mathematical structure, (*via* an interpretation function). The model is *sound* when two terms that are equivalent *w.r.t.* the evaluation rules are interpreted by the same denotation in the mathematical structure. All programs that return the same result for a given input are thus identified. Typically, two programs implementing different list-sorting algorithms will not be distinguished in the semantics (at least in the typed case). This way, it is possible to ignore syntax considerations and to focus on the *extensional properties* of the programs of a certain language (what is interesting is then *what* they compute, not *how*).

With denotational models, one can for example compare the expressive power of different languages, or determine whether one rule of a calculus is redundant (it does not calculate new values), or whether it is on the contrary *inconsistent* (by adding it to the calculus, all terms can reduce to the same result). For instance

Scott's domains [Sco70] (the first models for the untyped lambda calculus), have entailed the consistency of the *surjective pairing*[1] in the lambda calculus with pairs, though this rule is not derivable in the pure lambda calculus with pairs encoding.

Moreover, a denotational model may contain more objects than the denotations that are strictly necessary to interpret the terms of the language. It might be of interest to adopt the opposite approach: to start from the model and then to enrich the syntax. That is how the coherent spaces, a mathematical structure developed for a semantic study of the System F [GLT89, Chap. 14], led to the development of Linear Logic [Gir87].

Denotational models for the simply typed lambda calculus can be relatively simply constructed. In general a set model suffices: most of basic types α naturally correspond to a set (that is denoted by $|\alpha|$), with the terms of this type representing an element of this set. The type `nat` is for instance interpreted by the set of all integers, and each term of type `nat` actually corresponds to some integer. The type $A \to B$ is the interpreted by $|A \to B|$, the set of functions from $|A|$ to $|B|$. So the denotation for a lambda abstraction $\lambda x.t$ is a function mapping an element x of $|A|$ to the interpretation of t, and the denotation for an application tu relies on a fundamental property of the simply typed lambda calculus: if tu is of type A, then t necessarily has a type of the form $B \to A$, with u of type B:

$$\frac{t : B \to A \quad u : B}{tu : A}$$

Using an inductive reasoning, the denotation of t is an element of $|B \to A|$. Hence it is a function f from $|B|$ to $|A|$. Similarly, u is interpreted as an element e of $|B|$. So we can use the term $f(e)$ as a denotation for tu.

On the other hand, things are not as simple in the pure lambda calculus: every term can be an argument for any other term. This means that the mathematical object D in which terms are interpreted has to be the same (or at least isomorphic to) the functions space from D to D (that is written D^D). Hence the need of a mathematical object satisfying the recursive equation

$$D \;\cong\; D^D \;.$$

The only set that is a solution of this equation is *unit* (the set with only one element). Such a model would bring absolutely no information on the calculus, as all terms would receive the same denotation.] That is why some more constrained structures are required. The first construction solving this equation was proposed by Scott [Sco70].

Outline of the Thesis

In this manuscript we first present the lambda calculus with constructors as it was introduced in [AMR06]. It is based on a language that includes (constant)

[1] Given a construction (t_1, t_2) for the pair, and two primitive projections π_1, π_2 with rewriting rules $\pi_i(t_1, t_2) \to t_i$, the surjective pairing rule is $(\pi_1(t), \pi_2(t)) \to t$.

constructors, a pattern matching construction, and a *Daimon* in addition to the usual constructions of the lambda calculus. The Daimon is a constant with some specific reduction rules, that behaves like the `exit` instruction of some programming languages. The calculus has nine reduction rules, three of which are computationally essential (including the usual rule of β-reduction). An additional one is necessary for the confluence and two other ones (that may be called *administrative* rules) are only needed for the separation property (one these administrative rules is the usual η-reduction). The last three rules describe the interaction of the Daimon with the evaluation context. The Daimon and its reduction rules can be removed from the calculus, but they are sometimes useful to express or prove some properties of the lambda calculus with constructors.

We then describe a polymorphic type system for this calculus, with a subtyping order. This type system takes up a challenge set in [AMR06], since one of the essential rules of the calculus was presented there as ill-typed. It is a very expressive type system, that includes union and intersection operators, as well as second order existential and universal quantification. However it does not ensure strong normalisation of typable terms, and it does not prevent match failure at the execution. This is due to one of the administrative rules. Moreover subject reduction is not proved (union types raise some problems).

In Chap. 3 we overcome this weakness thanks to a realisability model for the typed calculus without the problematic administrative rule. This model ensures strong normalisation and absence of match failure in the restricted calculus. Furthermore it guarantees that typable terms actually reduce on values, even without the rule that we left out.

In Chap. 4, we then focus on denotational models for the untyped lambda calculus with constructors. We first define a class of Cartesian closed categories, that we call the $\lambda_{\mathscr{C}}$-models, and we detail a sound interpretation of the lambda calculus with constructors in such categories. We then show a completeness result for these models (in the sense that the definition does not require the identification of two terms which are not equivalent in the calculus—unless they raise a match failure). Completeness is established using a syntactic model of the calculus (the PER model, of Partial Equivalence Relations).

In the last chapter, we translate the lambda calculus with constructors into the lambda calculus with pairs, whose denotational semantics in Cartesian closed categories is well known. This translation is based on *Continuation Passing Style* techniques (or CPS [Plo75]). By transposing the structure of this translation at the level of models, we get a transformation of every continuation model [RS98] of the pure lambda calculus into a model of lambda calculus with constructors (*i.e.* a $\lambda_{\mathscr{C}}$-model, as defined in Chap. 4). This enables the construction of non-syntactic models of the lambda calculus with constructors.

Chapter 2

Typed lambda-calculus with constructors

2.1 The lambda calculus with constructors

2.1.1 Informal presentation

The lambda calculus with constructors [AMR09] extends the pure lambda calculus with pattern matching features: a countable set of constants c, c' *etc.* called *constructors* is added, with a simple mechanism of case analysis on these constants (similar to the `case` instruction of Pascal):

$$\{\!|\mathsf{c}_1 \mapsto t_1; \ldots; \mathsf{c}_k \mapsto t_k|\!\} \cdot \mathsf{c}_i \quad \to \quad t_i \qquad \text{(CaseCons)}$$

This enables a basic implementation of enumerated types. For instance, Booleans are represented using two constructors t and f, and by setting:

$$\texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \quad = \quad \{\!|\mathsf{t} \mapsto t_1; \mathsf{f} \mapsto t_2|\!\} \cdot b\ .$$

The case analysis allows actually to reduce

$$(\texttt{if}\ \mathsf{t}\ \texttt{then}\ t_1\ \texttt{else}\ t_2) \ \to\ t_1, \quad \text{and} \quad (\texttt{if}\ \mathsf{f}\ \texttt{then}\ t_1\ \texttt{else}\ t_2) \ \to\ t_2.$$

Although only constant constructors can be analysed, a matching on variant constructors can be performed *via* a commutation rule between case construction and application:

$$\{\!|\theta|\!\} \cdot (tu) \quad \to \quad (\{\!|\theta|\!\} \cdot t)\ u \qquad \text{(CaseApp)}$$

To understand how it works, consider the predecessor function (over unary integers) that maps $\mathsf{0}$ to $\mathsf{0}$, and $\mathsf{S}\ n$ to n. In the lambda calculus with constructors this function is implemented as

$$\texttt{pred} = \lambda n.\{\!|\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda x.x|\!\} \cdot n\ .$$

Applying this function to $\mathsf{0}$ or to $\mathsf{S}n$ produces effectively the expected result:

$$\begin{array}{rcl} \texttt{pred}\ \mathsf{0} & \to & \{\!|\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda x.x|\!\} \cdot \mathsf{0} \ \to\ \mathsf{0} \\ \texttt{pred}\ (\mathsf{S}\ n) & \to & \{\!|\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda x.x|\!\} \cdot (\mathsf{S}\ n) \\ & \to & (\{\!|\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda x.x|\!\} \cdot \mathsf{S})\ n \ \to\ (\lambda x.x)\ n \ \to\ n \end{array}$$

The head occurrence of constructor S is captured by the case construct *via* the reduction rule CaseApp, and its argument is bounded with a lambda abstraction in the branch of the case analyser. In fact, the whole expressiveness of ML-style pattern matching is reached with this mechanism:

In ML :	In the λ-calculus with constructors :

`match x with`
`  | c1 x1 ··· xn -> r1`
`  ⋮`
`  | ck y1 ··· ym -> rk`
	$\left\{\left\| \begin{array}{rcl} \mathsf{c}_1 & \mapsto & \lambda x_1 \dots x_n.r_1 \\ & \vdots & \\ \mathsf{c}_k & \mapsto & \lambda y_1 \dots y_m.rk \end{array} \right\|\right\} \cdot x$

Moreover, such a decomposition of pattern matching into a case analysis on constants and a commutation rule allows the use of variadic constructors (*i.e.* with no fixed arity).

Example 2.1.1 . We can chose a constructor $\mathsf{c}_\diamond$ for initialising arrays (with any number of elements). Then the function on arrays permuting the first two elements is written

$$\texttt{perm2} = \lambda z.\{\!|\mathsf{c}_\diamond \mapsto \lambda xy.\mathsf{c}_\diamond yx|\!\} \cdot z$$

and if $a = \mathsf{c}_\diamond\ t_1 \dots t_k$ represents an array with k elements ($k \geq 2$), then

$$\texttt{perm2}(a) \to^* \mathsf{c}_\diamond\ t_2 t_1 t_3 \dots t_k \ :$$

$$\begin{array}{rcll} \texttt{perm2}\ (a) & \to & \{\!|\mathsf{c}_\diamond \mapsto \lambda xy.\mathsf{c}_\diamond yx|\!\} \cdot (\mathsf{c}_\diamond\ t_1 \dots t_k) & (\textsc{AppLam}) \\ & \to^k & (\{\!|\mathsf{c}_\diamond \mapsto \lambda xy.\mathsf{c}_\diamond yx|\!\} \cdot \mathsf{c}_\diamond)\ t_1 \dots t_k & (\textsc{CaseApp}) \\ & \to & (\lambda xy.\mathsf{c}_\diamond yx)\ t_1 t_2 \dots t_k & (\textsc{CaseCons}) \\ & \to^2 & \mathsf{c}_\diamond t_2 t_1 t_3 \dots t_k & (\textsc{AppLam}) \end{array}$$

Finally there is also a special constant: the *Daimon* ✠. It is inherited from ludics [Gir01], and means immediate termination.

2.1.2 Syntax

Terms of the lambda calculus with constructors, or $\lambda_{\mathscr{C}}$-calculus, are defined from two disjoints sets of symbols: *variables* (denoted by x, y, z *etc.*) and *constructors* (written with typewriter letters $\mathsf{c}, \mathsf{d}, \mathsf{c}_1, \mathsf{c}_2$ *etc.*). The set $\mathscr{C}$ of constructors is countable and possibly infinite.

The syntax of the $\lambda_{\mathscr{C}}$-calculus consists of two syntactic classes, defined by mutual induction: *terms* (denoted by $s, t.u, v$ *etc.*) and *case-bindings* (written with Greek letters θ, ϕ, ψ *etc.*).

Definition 2.1.1 (*Syntax of the $\lambda_{\mathscr{C}}$-calculus*)

$$\begin{array}{rcll} s,t,u,v & := & x \mid tu \mid \lambda x.t \mid \mathsf{c} \mid \{\!|\theta|\!\} \cdot t \mid \maltese & \\ \theta,\phi & := & \{\mathsf{c}_1 \mapsto u_1; \ldots; \mathsf{c}_n \mapsto u_n\} & (\text{with } n \geq 0 \text{ and } \mathsf{c}_i \neq \mathsf{c}_j \text{ for } i \neq j) \end{array}$$

A case-binding is just a finite (partial) function from constructors to terms whose domain is written $\mathrm{dom}(\theta)$. By analogy with sequential notation, we may write θ_c for u when $\mathsf{c} \mapsto u \in \theta$. In order to ease the reading, we may write $\{\!|\mathsf{c}_1 \mapsto u_1; \ldots; \mathsf{c}_n \mapsto u_n|\!\} \cdot t$ instead of $\{\!|\{\mathsf{c}_1 \mapsto u_1; \ldots; \mathsf{c}_n \mapsto u_n\}|\!\} \cdot t$.

Terms include the constructs of the lambda calculus, that have the usual interpretation: application and λ-abstraction. A term can also be a constructor. As well as any term, constructors can be applied to any number of arguments, and thereby have no fixed arity. We call a *data-structure* a term on the form $\mathsf{c}t_1 \ldots t_n$. The *case construct* $\{\!|\theta|\!\} \cdot t$ represents the matching of t according to θ.

Writing conventions We extend the parenthesising conventions of [Bar84]: application has priority over abstraction and case construct, and parentheses associate on the left. Also the abstraction over several variables can be written with only one lambda. For instance:

$$\begin{array}{rcl} \lambda x.\mathsf{c}\ \mathsf{d}\ x & \text{means} & \lambda x.\big((\mathsf{c}\ \mathsf{d})\ x\big) \\ \lambda xy.\{\!|\theta|\!\} \cdot x\ \maltese & \text{means} & \lambda x.\lambda y.(\ \{\!|\theta|\!\} \cdot (x\ \maltese)) \end{array}$$

Variables and substitution We write $\mathrm{fv}(t)$ (and $\mathrm{fv}(\theta)$) the set of *free variables* of term t (*resp.* of case-binding θ). It is inductively defined in the standard way, given that the λ-abstraction is the only variable binder.

$$\begin{array}{rlrl} \mathrm{fv}(x) & = \{x\} & \mathrm{fv}(t\,u) & = \mathrm{fv}(t) \cup \mathrm{fv}(u) \\ \mathrm{fv}(\mathsf{c}) & = \emptyset & \mathrm{fv}(\{\!|\theta|\!\} \cdot t) & = \mathrm{fv}(\theta) \cup \mathrm{fv}(t) \\ \mathrm{fv}(\maltese) & = \emptyset & & \\ \mathrm{fv}(\lambda x.t) & = \mathrm{fv}(t) \setminus \{x\} & \mathrm{fv}(\theta) & = \bigcup_{\mathsf{c} \mapsto u \in \theta} \mathrm{fv}(u) \end{array}$$

As usual, the occurrences of a variable that appear non free in a term are said to be *bound*. We call a *closed term* a term with no free variables. The set of closed terms is written Λ_o.

We consider terms up to α-equivalence [Kri93, Chap. 1.2], *i.e.* we can freely rename all the bound occurrences of a variable in a term. In particular, we consider that terms do not have free and bound occurrences of the same variable. Notice that constructors are not variables, and thereby not subject to α-conversion nor substitution.

Substitution $t[x := u]$ (or $\theta[x := u]$) is defined as usual, by replacing all free occurrences of the variable x by the term u in t (*resp.* in θ).

2.1.3 Operational semantics

Reduction rules of the lambda calculus with constructors are named according to the constructs that are involved in the reduction. For instance, the usual β-reduction, performing the substitution of the bound variable in the body of a function by its argument, is now called AppLam, since it describes the interaction between an application and a λ-abstraction:

$$(\lambda x.t)\ u \quad \twoheadrightarrow \quad t[x := u] \qquad \text{(AppLam)}$$

In the same way, the standard η-reduction rule is now called LamApp:

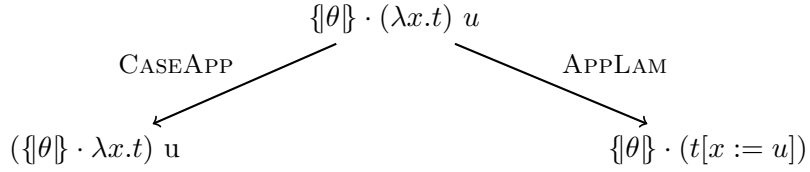
$$\text{If } x \notin \mathrm{fv}(t), \qquad \lambda x.t\,x \quad \twoheadrightarrow \quad t \qquad \text{(LamApp)}$$

This rule is computationally useless in the sense that it is not necessary to compute values, but it is necessary for the separation property (*cf.* Sec. 2.1.5).

In addition to these two rules of the usual lambda calculus, there are seven rules, specific to the new constructions of the lambda calculus with constructors. The whole system is given in Fig. 2.1.

Notations: The closure of the reduction relation $\twoheadrightarrow$ by context is denoted by $\to$. When $t \twoheadrightarrow t'$ with a rule R, we say that t is a R-redex, and we say that t' is a *reduct* of t (in one step) if $t \to t'$. We also may write $t \to_R t'$ to indicate that t reduces to t' with rule R, and $t \to^n t'$ (or $t \to^* t'$, or $t \to^= t'$) when t reduces on t' with n (*resp.* any number of, *resp.* zero or one) steps. A term is *in normal form* (or *normal*, for short) if it has no reduct, and it is *strongly normalising* if it necessarily reduces on a normal form (*i.e.* if it has no infinite reduction).

Case rules As we have seen in the presentation (Sec. 2.1.1), the key feature of the $\lambda_{\mathscr{C}}$-calculus is to decompose pattern matching into a case analysis on constant constructors (rule CaseCons) and a commutation rule between case construct and application (rule CaseApp). However this rule induces a critical pair:



This critical pair is closed by a commutation rule between case construct and λ-abstraction:

$$\text{If } x \notin \mathrm{fv}(\theta), \qquad \{|\theta|\} \cdot \lambda x.t \quad \twoheadrightarrow \quad \lambda x.\{|\theta|\} \cdot t \qquad \text{(CaseLam)}$$

With this rule, it is possible to reduce (considering $x \notin \mathrm{fv}(\theta)$ up to α-conversion)

$$\{\!|\theta|\!\} \cdot (\lambda x.t)\ u \quad \to \quad (\lambda x.\{\!|\theta|\!\} \cdot t)\ u \quad \to \quad \{\!|\theta|\!\} \cdot t\ [x := u] \ = \ \{\!|\theta|\!\} \cdot (t[x := u])$$

There is also a rule of commutation between two case constructs:

$$\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t \quad \twoheadrightarrow \quad \{\!|\theta \circ \phi|\!\} \cdot t \qquad \text{(CaseCons)}$$

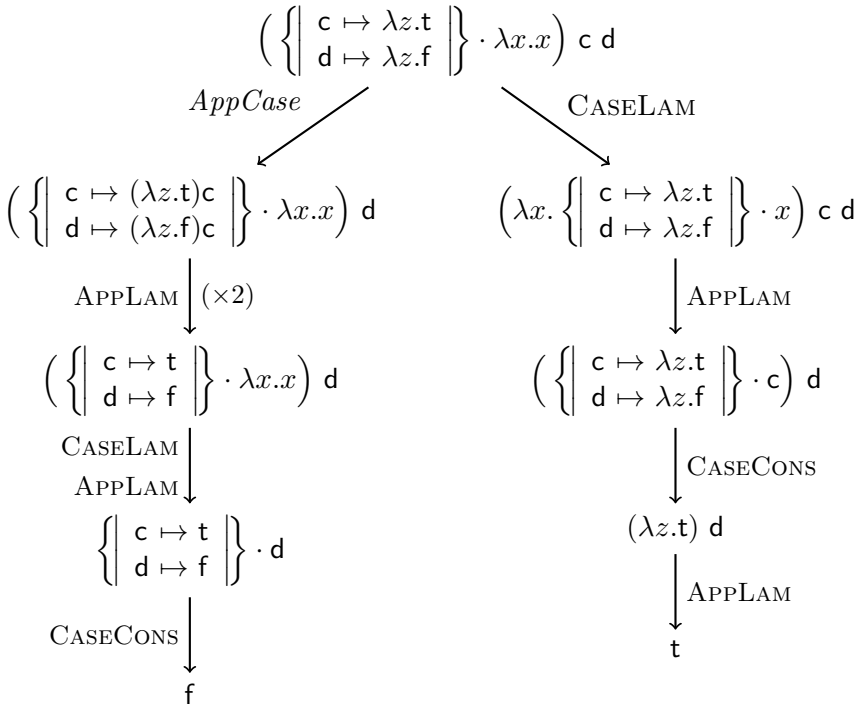
where the composition of two case-bindings $\theta \circ \phi$ means that the first one is replicated in each branch of the second:

$$\theta \circ \{\mathsf{c}_1 \mapsto u_1; \cdots \mathsf{c}_n \mapsto u_n\} \ = \ \{\mathsf{c}_1 \mapsto \{\!|\theta|\!\} \cdot u_1; \cdots \mathsf{c}_n \mapsto \{\!|\theta|\!\} \cdot u_n\}\ .$$

Remark 2.1.2 ($\lambda_{\mathscr{C}}$-calculus and commutative conversions) . Notice that neither CaseApp nor CaseLam corresponds to the commuting conversions coming from logic [GLT89, Sec. 10.4]. Indeed, a commuting conversion in our calculus would amount to pushing the elimination context inside the case-binding. For instance, the commutative conversion between case construct and application would be

$$(\{\!|\mathsf{c}_1 \mapsto u_1; \cdots ; \mathsf{c}_n \mapsto u_n|\!\} \cdot t)\ s \quad \twoheadrightarrow \quad \{\!|\mathsf{c}_1 \mapsto u_1 s\,; \cdots ; \mathsf{c}_n \mapsto u_n s\,|\!\} \cdot t$$

Such a rule (let us call it *AppCase*) is not compatible with $\lambda_{\mathscr{C}}$-calculus, because it would lead to unifying any constructors t, f (and more generally any terms), as follows.



From the point of view of typing, commutative conversions are more intuitive. In rule *AppCase*, $\big((\{\!|\mathsf{c}_1 \mapsto u_1; \cdots ; \mathsf{c}_n \mapsto u_n|\!\}\cdot t)\ s \ \to\ \{\!|\mathsf{c}_1 \mapsto u_1 s\,; \cdots ; \mathsf{c}_n \mapsto u_n s\,|\!\}\cdot t \big)$ the term t has to be a matchable value in both sides, and the result of its matching (one of the u_i's) is a function taking s as argument. Although in rule CaseApp, $\big(\{\!|\theta|\!\} \cdot ts \ \to\ (\{\!|\theta|\!\} \cdot t)\ s \big)$, the term t behaves as a function in the l.h.s. while it is seen as a matchable value in the r.h.s. We will see how to treat this paradox in Sec. 2.2.

On the other hand, rule CaseCase (where the external case-binding is duplicated in each branch of the head-position case-binding) is similar to the commutative conversions of logic. ∎

Daimon rules When it appears in head position of a term, the Daimon progressively destroys the whole term: a lambda abstraction is destroyed with rule LamDai, an application with AppDai and a case construct with CaseDai. Such a term always has ✠ as normal form. Thereby we can see the Daimon as the "`exit`" instruction in programming, that clears the evaluation context and terminates the program.

Since the Daimon cannot appear during a reduction if it is not in the initial term, it is possible to consider a sub-calculus of $\lambda_{\mathscr{C}}$ with no Daimon and no Daimon rule. However it is useful to express the separation property (Sec. 2.1.5), and we will also use it to prove a strong normalisation theorem (Sec. 3.2).

Beta-reduction

AppLam	(AL)	$(\lambda x.t)u$	$\to$	$t[x := x]u$	
AppDai	(AD)	✠ u	$\to$	✠	

Eta-reduction

LamApp	(LA)	$\lambda x.tx$	$\to$	t	$(x \notin \mathrm{fv}(t))$
LamDai	(LD)	$\lambda x.$✠	$\to$	✠	

Case propagation

CaseCons	(CO)	$\{\!|\theta|\!\} \cdot \mathsf{c}$	$\to$	t	$((\mathsf{c} \mapsto t) \in \theta)$
CaseDai	(CD)	$\{\!|\theta|\!\} \cdot$ ✠	$\to$	✠	
CaseApp	(CA)	$\{\!|\theta|\!\} \cdot (tu)$	$\to$	$(\{\!|\theta|\!\} \cdot t)u$	
CaseLam	(CL)	$\{\!|\theta|\!\} \cdot \lambda x.t$	$\to$	$\lambda x.\{\!|\theta|\!\} \cdot t$	$(x \notin \mathrm{fv}(\theta))$

Case composition

CaseCase (CC) $\quad \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t \ \to\ \{\!|\theta \circ \phi|\!\} \cdot t$

with $\theta \circ \{\mathsf{c}_1 \mapsto t_1; ...; \mathsf{c}_n \mapsto t_n\} = \{\mathsf{c}_1 \mapsto \{\!|\theta|\!\} \cdot t_1; ...; \mathsf{c}_n \mapsto \{\!|\theta|\!\} \cdot t_n\}$

Figure 2.1: Reduction rules for $\lambda_{\mathscr{C}}$.

2.1.4 Values and defined terms

In pure the lambda calculus, a *value* is a function (*i.e.* a λ-abstraction). In $\lambda_{\mathscr{C}}$-calculus, it can also be a data-structure. Also the set $\mathcal{V}$ of values is given by:

$$\mathcal{V} \;=\; \{\lambda x.t \,/\, \mathrm{t} \text{ is a } \lambda_{\mathscr{C}}\text{-term } \} \;\cup\; \{\mathsf{c}t_1 \dots t_k \,/\, \mathsf{c} \in \mathscr{C} \text{ and } t_1, \dots t_k \text{ are } \lambda_{\mathscr{C}}\text{-terms } \}.$$

A match failure is a term of the form $\{\!|\theta|\!\} \cdot \mathsf{c}$ such that $\mathsf{c} \notin \mathrm{dom}(\theta)$. We say that a term is *undefined* when one of its sub-term is a match failure, and that it is *defined* otherwise. A term whose all reducts (in any number of steps) are defined is said to be *hereditarily defined*. This notion was introduced in [AMR09] in order to express the separation property (*cf.* Sec. 2.1.5).

The set of irreducible defined terms is included in values (with the exception of ✠).

Proposition 2.1.1. *Every defined closed normal term is either ✠ or a value.*

Proof : *Let t be a closed defined term in normal form. By structural induction on t, we show that t is either ✠ or $\lambda x.t_0$ or $\mathsf{c}\, t_1 \dots t_k$ for some constructor c, and some terms t_i. Since t is closed, it is not a variable. If it is a constructor, the Daimon or an abstraction, the result holds. If it is an application, write $h\, t_1 \dots t_k = t$, where h is not an application. Then h is necessarily closed, defined and normal. It is not an abstraction, nor the Daimon (otherwise t would be reducible with* AppLam *or* AppDai*). Hence it is a data-structure by induction hypothesis, and so is t.*
Now assume $t = \{\!|\theta|\!\} \cdot h$. Then h is closed too, defined and normal. By induction hypothesis it is a value or the Daimon. It cannot be the Daimon, nor an abstraction, nor an application, otherwise t would be reducible with CaseDai, CaseLam *or* CaseApp*. So h is a constructor. If it is in the domain of θ, then t is reducible with* CaseCons*, and if it is not in the domain, t is not defined. Finally t cannot be a case construct.* □

Notice that the proof does not use rule CaseCase (and rules LamApp, LamDai neither), so the proposition holds for normal forms w.r.t. $\lambda^{-}_{\mathscr{C}}$-calculus.

Finally, a term which is both strongly normalising and hereditarily defined is said to be *perfectly normalising*. Perfect normalisation satisfies this usual lemma of lambda-calculus:

Lemma 2.1.2. *If $t[x := u]$ is perfectly normalising, then so is t.*

Proof : *First if $t \to t'$ then $t[x := u] \to t'[x := u]$ [AMR09, Lem. 9]. Thus, if $t[x := u]$ is strongly normalising, so is t. Then, if $t[x := u]$ is defined, it has no sub-term of the form $\{\!|\theta|\!\} \cdot \mathsf{c}$ with $\mathsf{c} \notin \mathrm{dom}(\theta)$, and this property is preserved by replacing some sub-terms u by x. So t is defined too. By induction on the reduction of t, we can easily conclude that if $t[x := u]$ is hereditarily defined, then so is t.* □

In the next part, we present a type system that will, to some extent, select only perfectly normalising terms.

2.1.5 Properties of the untyped calculus

In this section, we briefly recall some important properties that have been established in [AMR09].

Confluence

The confluence or non confluence of each sub-calculus of the lambda calculus has been proved in [AMR09, Theo. 1]. et $\mathscr{S}$ be a subset of the nine reduction rules given in Fig. 2.1. Then $\mathscr{S}$ is a confluent system if it satisfies the six following conditions:

$$\begin{array}{rcl}
\textsc{AppLam} \in \mathscr{S} \ \wedge \ \textsc{LamDai} \in \mathscr{S} & \Rightarrow & \textsc{AppDai} \in \mathscr{S} \\
\textsc{LamApp} \in \mathscr{S} \ \wedge \ \textsc{AppDai} \in \mathscr{S} & \Rightarrow & \textsc{LamDai} \in \mathscr{S} \\
\textsc{CaseApp} \in \mathscr{S} \ \wedge \ \textsc{AppLam} \in \mathscr{S} & \Rightarrow & \textsc{CaseLam} \in \mathscr{S} \\
\textsc{CaseApp} \in \mathscr{S} \ \wedge \ \textsc{AppDai} \in \mathscr{S} & \Rightarrow & \textsc{CaseDai} \in \mathscr{S} \\
\textsc{CaseLam} \in \mathscr{S} \ \wedge \ \textsc{LamApp} \in \mathscr{S} & \Rightarrow & \textsc{CaseApp} \in \mathscr{S} \\
\textsc{CaseLam} \in \mathscr{S} \ \wedge \ \textsc{LamDai} \in \mathscr{S} & \Rightarrow & \textsc{CaseDai} \in \mathscr{S}
\end{array}$$

In particular the whole type system enjoys the Church-Rosser property. In this document, we may need the following sub-systems of reductions, that all are confluent:

- $\mathcal{B}_{\mathscr{C}}$ denotes the $\lambda_{\mathscr{C}}$-calculus without rule AppLam.
- $\lambda^{-}_{\mathscr{C}}$ denotes the lambda calculus with constructors without case-composition.
- $\to_{\mathcal{B}}$ denotes a reduction step by a rule that is neither CaseApp nor CaseLam nor CaseCase.
- $\rightharpoonup$ represents a reduction step with the rule CaseApp or CaseLam.
- $\to_{cc}$ is the reduction system with only the rule CaseCase.
- The $\lambda_{\mathscr{C}}$-calculus without Daimon and without rules LamDai, AppDai and CaseDai is confluent too.

Separation

Arbiser, Miquel and Ríos have also proved in [AMR09, Theo. 2] that two distinct defined normal forms are *weakly separable*. That is, there is a context that evaluates one of them on the Daimon, and the other one on a match failure.

Formally, an *evaluation context* for the lambda calculus with constructors is given by the following grammar:

$$E[\,] \ := \quad [\,]\ t_1 \dots t_k \quad | \quad \{\!|\theta|\!\} \cdot [\,]\ t_1 \dots t_k$$

Then, for any defined normal forms t and t', either $t = t'$ (up to α-conversion), or there exists an evaluation context $E[\,]$ such that

$$\begin{array}{lll}
E[t_1] \to^* \maltese & \text{and} & E[t_2] \text{ reduces on an undefined term,} \quad \text{or} \\
E[t_2] \to^* \maltese & \text{and} & E[t_1] \text{ reduces on an undefined term.}
\end{array}$$

Remark that the Daimon absorbs any evaluation context: $E[✠] \to^* ✠$. The same holds for undefined terms: if $\mathsf{c} \notin \mathrm{dom}(\theta)$, then for any evaluation context $E[\,]$, all reducts of $E[\{\!|\theta|\!\} \cdot \mathsf{c}]$ are undefined. That is informally why no evaluation context is able to separate $\{\!|\mathsf{c} \mapsto t|\!\} \cdot \mathsf{d}_1$ from $\{\!|\mathsf{c} \mapsto u|\!\} \cdot \mathsf{d}_2$, and the separation property does not hold for undefined terms.

Strong normalisation of $\mathcal{B}_{\mathscr{C}}$.

The untyped lambda calculus with constructors is not normalising since it contains the usual β-reduction rule. However, it is the only delicate rule for normalisation, since the $\mathcal{B}_{\mathscr{C}}$-calculus (the $\lambda_{\mathscr{C}}$-calculus without rule AppLam) is *strongly normalising* (that is, no term has an infinite reduction). This is proved in [AMR09, Prop. 1], by showing that the *structural measure* $s(\cdot)$ (inductively defined below) strictly decreases for each rule of $\lambda_{\mathscr{C}}$-calculus, except AppLam. Since this measure is positive, there is no infinite reduction.

Definition 2.1.2 (*Structural measure*)

$$\begin{array}{lll} s(x) = 1 & s(\lambda x.t) = s(t)+1 & s(\{\!|\theta|\!\} \cdot t) = s(t) \times (s(\theta)+2) \\ s(\mathsf{c}) = 1 & s(tu) = s(t)+s(u) & s(\theta) = \sum_{\mathsf{c} \in \mathrm{dom}(\theta)} s(\theta_{\mathsf{c}}) \\ s(✠) = 1 & & \end{array}$$

We shall later refer to the same measure to ensure that every sub-calculus of the $\lambda_{\mathscr{C}}$-calculus that does not contain rule AppLam is strongly normalising.

2.2 Type system

In natural languages some sentences are *grammatically* correct but *semantically* not meaningful:

"Le silence vertébral indispose la voile licite" [1].

This relative freedom is certainly essential to great poetic creations, but computer scientists usually prefer effectiveness to artistic beauty. That is to say, we do not only ask a program to be *syntactically* correct, we also want its evaluation to reach a *value* (that we can see as the *meaning* of the program).

For instance, we want to rule out the application of the function `perm2` (that permutes the first two elements of an array, *cf.* Ex. 2.1.1) to an integer $\mathsf{S}\ n$. More generally, we want to prevent match failure in the reduction. We consider that they are *incorrect* terms.

Most programming languages come with a notion of *correct* program. In order to restrict the set of programs that can be written, we can associate to the language a *type system*: the challenge is to design a term annotation method that

[1] Example given by the French linguist Lucien Tesnière [Tes53], that was translated by Noam Chomsky as "Colourless green ideas sleep furiously" [Cho57]... Unlike the sentence, we might say that the translation is semantically correct, but is syntactically not exactly faithful.

enables annotating (with types) as many programs as possible, but only correct ones. This is what is called the *correctness* of the type system. One might also be interested in its *completeness* ("is every correct program typable?") or in the *type inference* ("is there an algorithm that decides what is the type of a program, if it has some?").

Unfortunately in this thesis we only deal with the first question: we consider that a term has a well behaviour when its evaluation always terminates and does not lead to match failure. With this view we now define a type system for the lambda calculus with constructors, and we show in next chapter that it is correct, up to some restrictions we shall explain in Sec. 2.3.2.

2.2.1 Main ideas

We propose a polymorphic type system for the lambda calculus with constructors. It includes the simply typed λ-calculus: the main type construct is the arrow type $T \to U$, coming with its usual introduction and elimination rules for typing λ-abstraction and application respectively:

$$\frac{x : T \text{ implies } u : U}{\lambda x.u : T \to U} \qquad \frac{t : U \to T \text{ and } u : U}{tu : T}$$

Types for data-structures are modelled on the syntax of terms: a term $\mathsf{c}t_1 \dots t_k$ will have type $\mathbf{c}T_1 \dots T_k$ if each t_i has type T_i. So we associate to each constructor c a constant type $\mathbf{c}$ (written with bold font), and we introduce the notion of application type. At a first sight, the most natural way to introduce an application type would be:

$$\frac{t : T \text{ and } u : U}{tu : TU}$$

Yet it would allow typing "incorrect" application terms. For example a function waiting for Boolean arguments could receive integer arguments. Furthermore, since the term $\delta = \lambda x.xx$ is typable in polymorphic type systems (one of its possible type is $\Delta = (\forall X.X \to X) \to (\forall X.X \to X)$), the diverging term $\delta\delta$ would be typable with type $\Delta\Delta$. That is why we restrict the application type, and prevent a derivation such as

$$\frac{t : (U \to T) \text{ and } u : U'}{tu : (U \to T)U'}$$

To do so, we distinguish a subclass of types that we call *data-types*. It contains no arrow types, and it characterises the types of data-structures. Data-types are the only ones that we apply to other types.

Since constructors are variadic, any data-structure can be viewed as a function expecting one more argument. This is expressed by a sub-typing rule: any data-type D is a subtype of $T \to DT$. That is why the type system is provided with a sub-typing relation, and "T is a subtype of U" means that every term of type T has type U too.

Algebraic types are constructed with a union operator. Because of sub-typing, we also use its dual: intersection operator. Finally, polymorphism is achieved with type (and data-type) variables, and universal and existential quantification.

2.2.2 Type syntax

The lambda calculus with constructors is provided with a polymorphic type system with union and intersection operators. Types (notation: T, U) and data-types (notation: D, E) are defined by mutual induction in Fig. 2.2. Notice that quan-

$T, U :=$	X	(Ordinary type variable)
	$\mid \alpha \mid \mathbf{c} \mid DT$	(Data-type)
	$\mid T \to U$	(Arrow type)
	$\mid T \cup U$	(Union type)
	$\mid T \cap U$	(Intersection type)
	$\mid \forall \alpha.T \mid \forall X.T$	(Universal type)
	$\mid \exists \alpha.T \mid \exists X.T$	(Existential type)
$D, E :=$	α	(Data-type variable)
	$\mid \mathbf{c} \mid DT$	(Data-structure)
	$\mid D \cup E$	(Union data-type)
	$\mid D \cap E$	(Intersection data-type)
	$\mid \forall \alpha.D \mid \forall X.D$	(Universal data-type)
	$\mid \exists \alpha.D \mid \exists X.D$	(Existential data-type)

Figure 2.2: Syntax of Types

tification over data-types requires the use of two different kinds of type variables: capital letters from the end of Latin alphabet (X, Y etc.) denote ordinary type variables, whereas Greek letters from the beginning of the alphabet (α, β) denote data-type variables. We may write ν a variable that is either an ordinary or a data-type variable.

An arrow type $U \to T$ cannot be applied to another type. By analogy with term convention, type application takes precedence over all the other operators and is left associative, whereas arrow is right associative and has the less precedence. In between, union and intersection take precedence over universal and existential quantifier. So that:

$$\forall X.\mathbf{c}XT \to T \text{ means } \Big(\forall X.\big((\mathbf{c}X)T\big)\Big) \to T$$
$$\exists \alpha.\alpha X \cup \alpha T \to (X \to T) \to T \text{ means } \Big(\exists \alpha.\big((\alpha X) \cup (\alpha T)\big)\Big) \to \big((X \to T) \to T\big)$$

Free type variables and substitution

The set $\mathrm{TV}(T)$ of all free type variables of a type T is defined as expected:

$$\begin{array}{rlrlrl} \mathrm{TV}(X) = & \{X\} & \mathrm{TV}(\alpha) = & \{\alpha\} & \mathrm{TV}(\mathbf{c}) = & \emptyset \\ \mathrm{TV}(T \to U) = & \mathrm{TV}(T) \cup \mathrm{TV}(U) & & & \mathrm{TV}(DT) = & \mathrm{TV}(D) \cup \mathrm{TV}(T) \\ \mathrm{TV}(T \cap U) = & \mathrm{TV}(T) \cup \mathrm{TV}(U) & & & \mathrm{TV}(T \cup U) = & \mathrm{TV}(T) \cup \mathrm{TV}(U) \\ \mathrm{TV}(\forall \nu.T) = & \mathrm{TV}(T) \setminus \{\nu\} & & & \mathrm{TV}(\exists \nu.T) = & \mathrm{TV}(T) \setminus \{\nu\} \end{array}$$

Types also are considered up to renaming of bound variables, and the substitution of ν by a type U in T (written $T\{U/\nu\}$) only deals with *free* occurrences of ν in T.

Vectorial notation

In order to ease the reading, we may use a vectorial notation for terms and types. It is an adaptation of the telescopic mappings of de Bruijn [dB91] in a system with no dependent types but with type application. Intuitively, a type vector (or a term vector) is just a (possibly empty) sequence of types $\vec{T}$ (or a sequence of terms $\vec{u}$), and we can *telescope* many arrows or many application in one by writing $\vec{T} \to U$ or $D\vec{T}$ (or $t\vec{u}$):

$$\vec{T}, \vec{U} := [\,] \mid \vec{T}; U \qquad \vec{t}, \vec{u} := [\,] \mid \vec{t}; u$$

$$\begin{array}{rlrlrl} [\,] \to T & = T & & & D\,[\,] & = D & t\,[\,] & = t \\ (\vec{T}; U) \to T & = \vec{T} \to (U \to T) & & & D\,(\vec{T}; U) & = (D\vec{T})\,U & t(\vec{u}; u) & = (t\vec{u})u \end{array}$$

For convenience, we may write $T_1; \ldots; T_k$ instead of $[\,]; T_1; \ldots; T_k$ for non-empty vectors. For instance,

$$\mathbf{c}(U_1; U_2) \to (T_1; T_2) \to T' \quad \text{denotes} \quad \mathbf{c}U_1U_2 \to T_1 \to T_2 \to T'$$

We will see in next section that this notation makes sense with typing rules.

Remark 2.2.1 . Vectorial notation is not ambiguous: a type written with vectors denotes only one type of the original syntax. Conversely, a type can have many vectorial representations (when it has many successive arrows):

$$[\,] \to (U_1 \to U_2 \to T) \quad , \quad ([\,]; U_1) \to (U_2 \to T)\ , \quad \text{and } ([\,]; U_1; U_2) \to T$$

are three possible notations for $U_1 \to U_2 \to T$.

2.2.3 Typing and sub-typing rules

A typing judgement is on the form $\Gamma \vdash M : T$, where M is a term or a case-binding (the same syntax of types is used for both), and Γ is a context, *i.e.* an unordered sequence of couples $x : U$ assigning a type to a variable.

Typing rules are given in Fig. 2.3. They include the usual introduction and elimination rules of typed λ-calculus for each type operator. Some of them —like the elimination of universal quantifier— are indeed sub-typing rules (Fig. 2.4). The sub-typing relation, written with the infix notation $\preccurlyeq$, is a preorder and we write $\equiv$ the induced equivalence relation: $T \equiv U$ when $T \preccurlyeq U$ and $U \preccurlyeq T$

Typing variadic constructors

Atomic data-structures (the constructors) are typed with rule **Constr**:

$$\Gamma \vdash \mathsf{c} : \mathbf{c}.$$

Then, sub-typing rule **Data** allows to see this constructor as a function that can be applied to any typable term:

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{c} : \mathbf{c} \qquad \mathbf{c} \preccurlyeq T \to \mathbf{c}T}{\Gamma \vdash \mathsf{c} : T \to \mathbf{c}T} \qquad \Gamma \vdash t : T}{\Gamma \vdash \mathsf{c}t : \mathbf{c}T}$$

By iterating this derivation, $\mathsf{c}t_1 \dots t_k$ gets type $\mathbf{c}T_1 \dots T_k$ as soon as every t_i has type T_i. This way, we can type constructors with any number of arguments.

Remark 2.2.2 . Using vectorial notation, we can write a sub-typing rule that *generalises* rule **Data** and that is derivable:

$$D \preccurlyeq \vec{T} \to D\vec{T}\ .$$

Furthermore, if we write $\Gamma \vdash \vec{u} : \vec{U}$ when $\vec{u} = u_1; \cdots ; u_k$ and $\vec{U} = U_1; \cdots ; U_k$ and $\Gamma \vdash u_i : U_i$ for all $i \leq k$, then the following rule is derivable (by induction on the length k of the vectors):

$$\dfrac{\Gamma \vdash t : \vec{U} \to T \quad \Gamma \vdash \vec{u} : \vec{U}}{\Gamma \vdash t\vec{u} : T}$$

By combining these two "generalised" rules, $\Gamma \vdash \mathsf{c}\vec{t} : \mathbf{c}\vec{T}$ immediately follows from $\Gamma \vdash \vec{t} : \vec{T}$.

Sub-typing rule **Discr** expresses in the type system that different constructors can be discriminated. Indeed, if $\mathsf{c}_1 \neq \mathsf{c}_2$ then $\mathbf{c}_1\vec{T}$ and $\mathbf{c}_2\vec{U}$ cannot type the same term t, except if t also has type $\forall\alpha.a$. (which means that $t \to^* \maltese$ —*cf.* page 3.2.1, so it is not a data-structure).

Thus we can consider a union of data-types as a disjoint union if all constructor types in head position are different. That is why we call *algebraic type* a type on the form

$$\bigcup_{i \leq k} \mathbf{c}_i\, \vec{U_i} \qquad \text{with } \mathbf{c}_i \neq \mathbf{c}_j \quad \text{if } i \neq j.$$

Typing case-bindings

Types for case-bindings are the same as the ones for terms. A case-binding is typed (with rule **Cb**) like a function waiting for a constructor of its domain as argument, up to a possible conversion of arrow type into application type: from a typing judgement $\Gamma \vdash u : T \to U$, both following derivations are valid:

$$\frac{\Gamma \vdash u : U \to T}{\Gamma \vdash \{\mathsf{c} \mapsto u\} : \mathbf{c} \to U \to T}\,\mathbf{Cb} \qquad \frac{\Gamma \vdash u : U \to T}{\Gamma \vdash \{\mathsf{c} \mapsto u\} : \mathbf{c}U \to T}\,\mathbf{Cb}$$

That is because an arrow type can be decomposed in different ways, and thereby have several vectorial denotations (*cf.* remark 2.2.1). This is the point that allows CaseApp commutation rule to be well typed.

Example 2.2.3 (Ambiguity of rule **Cb**) . Again, consider the constructor $\mathsf{c}_\diamond$ that initialises arrays. Then the case-binding $\theta = \{\mathsf{c}_\diamond \mapsto \lambda xy.\mathsf{c}_\diamond x\}$ removes the second element of any array:

$$\{\!|\theta|\!\}\cdot(\mathsf{c}_\diamond t_1 t_2 t_3) \to^3_{\textsc{CaseApp}} (\{\!|\theta|\!\}\cdot\mathsf{c}_\diamond)t_1t_2t_3 \to_{\textsc{CaseCons}} (\lambda xy.\mathsf{c}_\diamond x)t_1t_2t_3 \to^2_{\textsc{AppLam}} \mathsf{c}_\diamond t_1 t_3$$

From $\vdash t_1 : T_1$, $\vdash t_2 : T_2$ and $\vdash t_3 : T_3$ we can derive $\vdash \{\!|\theta|\!\} \cdot (\mathsf{c}_\diamond t_1 t_2 t_3) : \mathbf{c}_\diamond T_1 T_3$:

$$\dfrac{\dfrac{\dfrac{}{\vdash \lambda xy.\mathsf{c}_\diamond x : T_1 \to T_2 \to \mathbf{c}_\diamond T_1} \qquad \dfrac{\dfrac{}{\mathbf{c}_\diamond T_1 \preccurlyeq T_3 \to \mathbf{c}_\diamond T_1 T_3}}{T_1 \to T_2 \to \mathbf{c}_\diamond T_1 \preccurlyeq T_1 \to T_2 \to T_3 \to \mathbf{c}_\diamond T_1 T_3}}{\vdash \lambda xy.\mathsf{c}_\diamond x : T_1 \to T_2 \to T_3 \to \mathbf{c}_\diamond T_1 T_3}}{\vdash \theta : \mathbf{c}_\diamond T_1 T_2 T_3 \to \mathbf{c}_\diamond T_1 T_3}\,\mathbf{Cb}$$

$$\dfrac{\dfrac{}{\vdash \theta : \mathbf{c}_\diamond T_1 T_2 T_3 \to \mathbf{c}_\diamond T_1 T_3} \qquad \dfrac{\begin{array}{c}\vdash t_1 : T_1 \\ \vdash t_2 : T_2 \\ \vdash t_3 : T_3\end{array}}{\vdash \mathsf{c}_\diamond t_1 t_2 t_3 : \mathbf{c}_\diamond T_1 T_2 T_3}}{\vdash \{\!|\theta|\!\} \cdot (\mathsf{c}_\diamond t_1 t_2 t_3) : \mathbf{c}_\diamond T_1 T_3}\,\mathbf{case}$$

We can also give the same type to $(\{\!|\theta|\!\} \cdot \mathsf{c}_\diamond)t_1t_2t_3$ by choosing another possible type for θ (we write $\vec{T} = T_1; T_2; T_3$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \lambda xy.\mathsf{c}_\diamond x : \vec{T} \to \mathbf{c}_\diamond T_1 T_3}\,\mathbf{Cb}}{\vdash \theta : \mathbf{c}_\diamond \to \vec{T} \to \mathbf{c}_\diamond T_1 T_3} \qquad \dfrac{}{\vdash \mathsf{c}_\diamond : \mathbf{c}_\diamond}\,\mathbf{case}}{\vdash \{\!|\theta|\!\} \cdot \mathsf{c}_\diamond : \vec{T} \to \mathbf{c}_\diamond T_1 T_3} \qquad \begin{array}{c}\vdash t_1 : T_1 \\ \vdash t_2 : T_2 \\ \vdash t_3 : T_3\end{array}}{\vdash (\{\!|\theta|\!\} \cdot \mathsf{c}_\diamond)t_1t_2t_3 : \mathbf{c}_\diamond T_1 T_3}$$

In order to give the same type to $\{\!|\theta|\!\} \cdot (\mathsf{c}_\diamond t_1 t_2 t_3)$ and $(\{\!|\theta|\!\} \cdot \mathsf{c}_\diamond)t_1t_2t_3$ we have chosen different vectorial denotations for the type $T_1 \to T_2 \to T_3 \to \mathbf{c}_\diamond T_1 T_3$ when applying the typing rule **Cb**.

In the same way, the typing rule (**case**) for a case construct $\{\!|\theta|\!\} \cdot t$ allows t to be a function that waits for an arbitrary number of arguments. This makes CaseLam well typed. Indeed, if a case-binding θ has type $T \to U$, then both terms $\{\!|\theta|\!\} \cdot \lambda x.x$ and $\lambda x.(\{\!|\theta|\!\} \cdot x)$ are typable with the same type:

$$\dfrac{\dfrac{x : T \vdash x : T \qquad x : T \vdash \theta : T \to U}{x : T \vdash \{\!|\theta|\!\} \cdot x : U}\ \textbf{case}}{\vdash \lambda x.(\{\!|\theta|\!\} \cdot x) : T \to U} \qquad \dfrac{\vdash \lambda x.x : T \to T \qquad \vdash \theta : T \to U}{\vdash \{\!|\theta|\!\} \cdot \lambda x.x : T \to U}\ \textbf{case}$$

However the typing judgement obtained by the derivation rule **case** is determined by the premises, which is not the case when rule **Cb** is applied.

Algebraic types

If the case-binding we want to type (say $\theta = \{\mathsf{c}_i \mapsto u_i / 1 \leq i \leq n\}$) includes many branches, we can either chose one of them (for instance $\mathsf{c}_j \mapsto u_j$) and apply only once rule **Cb**:

$$\dfrac{\left(\Gamma \vdash u_i : \vec{U_i} \to T_i\right)_{i=1}^{n}}{\Gamma \vdash \theta : \mathbf{c}_j \vec{U_j} \to T_j}$$

(Actually in the premises, we only require that u_j has type $\vec{U_j} \to T_j$ and that every u_i is typable in context Γ). If we do so, we can only type a case construct $\{\!|\theta|\!\} \cdot t$ with a term t of type $\vec{T} \to \mathbf{c}_j \vec{U_j}$ for some type vector $\vec{T}$. This means that the constructor expected in head position to perform pattern matching is c_j (again this will be formally established thanks to the denotational model). If we do not know in advance which constructor will be matched by θ we can give it all possible types, using intersection operator: $\Gamma \vdash \theta : \bigcap_{1 \leq i \leq n} (\mathbf{c}_i \vec{U_i} \to T_i)$. Since we need θ to have an arrow type in order to associate it to a term with typing rule **case**, we then commute the intersection with the arrow:

$$\dfrac{\Gamma \vdash \theta : \bigcap_{1 \leq i \leq n} (\mathbf{c}_i \vec{U_i} \to T_i) \qquad \bigcap_{1 \leq i \leq n} (\mathbf{c}_i \vec{U_i} \to T_i) \preccurlyeq (\bigcup_{1 \leq i \leq n} \mathbf{c}_i \vec{U_i}) \to (\bigcup_{1 \leq i \leq n} T_i)}{\Gamma \vdash \theta : (\bigcup_{1 \leq i \leq n} \mathbf{c}_i \vec{U_i}) \to (\bigcup_{1 \leq i \leq n} T_i)}$$

Then to type $\{\!|\theta|\!\} \cdot t$ we just need t to have the algebraic type $\bigcup_{1 \leq i \leq n} \mathbf{c}_i \vec{U_i}$, and the constructor that will be analysed by θ can be any constructor of its domain.

Example 2.2.4 (Typing multi-branches case-bindings) . Assume nat is a type satisfying $nat \equiv \mathbf{0} \cup \mathbf{S}\, nat$. The predecessor case-binding

$$\theta = \{\mathsf{0} \mapsto \mathsf{0}\ ; \mathsf{S} \mapsto \lambda x.x\}$$

has both types $\mathbf{0} \to nat$ and $\mathbf{S}\, nat \to nat$. Hence we can derive

$$\dfrac{\vdash \theta : (\mathbf{0} \to nat) \cap (\mathbf{S}\, nat \to nat) \quad (\mathbf{0} \to nat) \cap (\mathbf{S}\, nat \to nat) \preccurlyeq (\mathbf{0} \cup \mathbf{S}\, nat) \to nat}{\vdash \theta : \ (\mathbf{0} \cup \mathbf{S}\, nat) \to nat}$$

and thus θ has type $nat \to nat$.

Rule $\mathbf{Cb}_\bot$ is a kind of generalisation of this typing derivation: if $\theta = \{\mathsf{c}_i \mapsto u_i \,/\, 1 \leq i \leq n\}$, with $\vdash u_i : \vec{U_i} \to T_i$, then for any $J \subseteq [\![1..n]\!]$, the judgement $\vdash \theta : \bigcup_{i\in J} \mathbf{c}_i\vec{U_i} \to \bigcup_{i\in J} T_i$ is derivable. Taking $J = \emptyset$, this would be written $\vdash \theta : \forall\alpha.\alpha \to \forall X.X$, as $\forall\alpha.\alpha$ is the lower bound of data-types, and $\forall X.X$ the lower bound of types. In particular, $\mathbf{Cb}_\bot$ enables typing the empty case-binding. Notice that the only way to type a term $\{\!|\emptyset|\!\} \cdot t$ is that t has type $\forall\alpha.\alpha$, and this means that t is (or reduces on) the Daimon —we will see that this is a consequence of the denotational model at the end of Sec. 3.2.1 (page 45).

Terms:

$$\mathbf{Init}\frac{-}{\Gamma \vdash x : T}\ (x{:}T\in\Gamma) \qquad \mathbf{False}\frac{-}{\Gamma \vdash \maltese : T} \qquad \mathbf{Constr}\frac{-}{\Gamma \vdash \mathsf{c} : \mathbf{c}}$$

$$\to\mathbf{intro}\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x.t : U \to T} \qquad \to\mathbf{elim}\frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u : U}{\Gamma \vdash tu : T}$$

$$\mathbf{case}\frac{\Gamma \vdash t : \vec{U} \to T \qquad \Gamma \vdash \theta : T \to T'}{\Gamma \vdash \{\!|\theta|\!\} \cdot t : \vec{U} \to T'}$$

Case Binding: If $\theta = (\mathsf{c}_i \mapsto u_i)_{i=1}^n$,

$$\mathbf{Cb}_\bot\ \frac{\left(\Gamma \vdash u_i : T_i\right)_{i=1}^n}{\Gamma \vdash \theta : \forall\alpha.\alpha \to \forall X.X} \qquad \mathbf{Cb}\ \frac{\left(\Gamma \vdash u_i : \vec{U_i} \to T_i\right)_{i=1}^n}{\Gamma \vdash \theta : \mathbf{c}_{i_0}\vec{U_{i_0}} \to T_{i_0}}\ (1\leq i_0\leq n)$$

Shared rules: M is either a term t, either a case binding θ.

$$\mathbf{Univ}\frac{\Gamma \vdash M : T}{\Gamma \vdash M : \forall\nu.T}(\nu\notin \mathrm{TV}(\Gamma)) \qquad \mathbf{Inter}\frac{\Gamma \vdash M : T \quad \Gamma \vdash M : U}{\Gamma \vdash M : T \cap U}$$

$$\mathbf{Exist}\frac{\Gamma, x : T \vdash M : U}{\Gamma, x : \exists\nu.T \vdash M : U}(\nu\notin \mathrm{TV}(U)) \qquad \mathbf{Union}\frac{\Gamma, x : T_1 \vdash M : U \quad \Gamma, x : T_2 \vdash M : U}{\Gamma, x : T_1 \cup T_2 \vdash M : U}$$

$$\mathbf{Subs}\frac{\Gamma \vdash M : T \quad T \preccurlyeq U}{\Gamma \vdash M : U}$$

Figure 2.3: Typing rules

Commutation rules for type operators

Commutation rules between arrow and universal quantifier are usual in *System F* with sub-typing [Mit88]. Commutation with union and intersection are also well known [Pie91]. More generally, all these rules are guided by semantic intuitions:

Preorder relation:

$$\textbf{Refl}\frac{-}{T \preccurlyeq T} \qquad \textbf{Trans}\frac{T \preccurlyeq T_0 \quad T_0 \preccurlyeq T'}{T \preccurlyeq T'}$$

Monotonicity and contravariance:

$$\textbf{Arrow}\frac{U' \preccurlyeq U \quad T \preccurlyeq T'}{U \to T \preccurlyeq U' \to T'} \qquad \textbf{App}\frac{D \preccurlyeq D' \quad T \preccurlyeq T'}{DT \preccurlyeq D'T'}$$

Elimination and introduction rules:

$$\cap\textbf{intro}\frac{T \preccurlyeq U_1 \quad T \preccurlyeq U_2}{T \preccurlyeq U_1 \cap U_2} \qquad \cap\textbf{elimL}\frac{-}{U_1 \cap U_2 \preccurlyeq U_1} \qquad \cap\textbf{elimR}\frac{-}{U_1 \cap U_2 \preccurlyeq U_2}$$

$$\cup\textbf{introL}\frac{-}{U_1 \preccurlyeq U_1 \cup U_2} \qquad \cup\textbf{introR}\frac{-}{U_2 \preccurlyeq U_1 \cup U_2} \qquad \cup\textbf{elim}\frac{T_1 \preccurlyeq U \quad T_2 \preccurlyeq U}{T_1 \cup T_2 \preccurlyeq U}$$

$$\forall\textbf{intro}\frac{T \preccurlyeq U}{T \preccurlyeq \forall\nu.U}{\scriptstyle(\nu \notin \mathrm{TV}(T))} \qquad \forall\textbf{elim}\frac{-}{\forall X.T \preccurlyeq T\{U/X\}} \qquad \forall\textbf{elimD}\frac{-}{\forall\alpha.T \preccurlyeq T\{D/\alpha\}}$$

$$\exists\textbf{intro}\frac{-}{T\{U/X\} \preccurlyeq \exists X.T} \qquad \exists\textbf{introD}\frac{-}{T\{D/\alpha\} \preccurlyeq \exists\alpha.T} \qquad \exists\textbf{elim}\frac{U \preccurlyeq T}{\exists\nu.U \preccurlyeq T}{\scriptstyle(\nu \notin \mathrm{TV}(T))}$$

$$\textbf{Data}\frac{-}{D \preccurlyeq T \to DT} \qquad \textbf{Discr}\frac{-}{\mathbf{c}_1\vec{T} \cap \mathbf{c}_2\vec{U} \preccurlyeq \forall\alpha.\alpha}\ {\scriptstyle(\mathbf{c}_1 \neq \mathbf{c}_2)}$$

Commutation between type operators:

$$\textbf{App}/\cap\frac{-}{DT \cap D'T' \preccurlyeq (D \cap D')(T \cap T')} \qquad \textbf{App}/\forall\frac{-}{\forall\nu.(DT) \preccurlyeq (\forall\nu.D)(\forall\nu.T)}$$

$$\to/\cap\frac{-}{(U \to T) \cap (U' \to T') \preccurlyeq (U \cap U') \to (T \cap T')} \qquad \to/\forall\frac{-}{\forall\nu.(T \to U) \preccurlyeq (\forall\nu.T) \to (\forall\nu.U)}$$

$$\to/\cup\frac{-}{(U \to T) \cap (U' \to T') \preccurlyeq (U \cup U') \to (T \cup T')} \qquad \to/\exists\frac{-}{\forall\nu.(T \to U) \preccurlyeq (\exists\nu.T) \to (\exists\nu.U)}$$

$$\cap/\cup\frac{-}{(U \cup T) \cap (U \cup T') \preccurlyeq U \cup (T \cap T')}$$

$$\cup/\textbf{AppR}\frac{-}{D(T \cup T') \preccurlyeq DT \cup DT'} \qquad \cup/\textbf{AppL}\frac{-}{(D \cup D')T \preccurlyeq DT \cup DT'}$$

$$\exists/\textbf{AppR}\frac{-}{D(\exists\nu.T) \preccurlyeq \exists\nu.DT}{\scriptstyle(\nu \notin \mathrm{TV}(D))} \qquad \exists/\textbf{AppL}\frac{-}{(\exists\nu.D)T \preccurlyeq \exists\nu.DT}{\scriptstyle(\nu \notin \mathrm{TV}(T))}$$

$$\cup/\forall\frac{-}{\forall\nu.(T \cup U) \preccurlyeq (\forall\nu.T) \cup U}{\scriptstyle(\nu \notin \mathrm{TV}(U))} \qquad \exists/\cap\frac{-}{(\exists\nu.T) \cap U \preccurlyeq \exists\nu.(T \cap U)}{\scriptstyle(\nu \notin \mathrm{TV}(U))}$$

Figure 2.4: Sub-typing rules.

terms must have type $T \cap U$ (*resp.* $T \cup U$) *if and only if* they have both (*resp.* one of) types T and U. The idea is the same for quantifiers (universal quantification is seen as a generalised intersection, and existential quantification as a generalised union). Concerning arrow types, a term must have type $U \to T$ *iff* we can apply it to any argument of type U and get a term of type T. This explains for instance the commutation rule between intersection and arrow:

$$(U_1 \to T_1) \cap (U_2 \to T_2) \preccurlyeq (U_1 \cup U_2) \to (T_1 \cup T_2) \ .$$

If a term can be applied to a term of type U_1 to form a term of type T_1, and also to a term of type U_2 to form a term of type T_2, then we can apply it to a term that has type U_1 or U_2 and we will get a term of type T_1 or of type T_2. Notice that this meaning of types also validates the rule

$$(U_1 \to T_1) \cap (U_2 \to T_2) \preccurlyeq (U_1 \cap U_2) \to (T_1 \cap T_2) \ .$$

The new construction of this type system, namely type application, is interpreted in a very naive way: a term has type DT if it is the application of a sub-term of type D and a sub-term of type T. This is why the rule $D(T_1 \cup T_2) \preccurlyeq DT_1 \cup DT_2$ is valid, but not $(D_1 \cup D_2)(T_1 \cup T_2) \preccurlyeq D_1T_1 \cup D_2T_2$.

2.3 Some non-properties of the typed $\lambda_{\mathscr{C}}$-calculus

2.3.1 Discussion on Subject reduction

Union types, seen as the dual of intersection types, are known to be problematic *w.r.t.* subject reduction in the lambda calculus. Different formalisms exist for the elimination rule of union: we use a left rule, inspired from sequent calculus, but a right elimination rule is also often used [Pie91, FP91, Rib07a]. With a cut rule, both presentations are equivalent [BDCd95, Theo. 1.5] (and rule **case** can be seen as a cut-rule for algebraic types).

However the same problem always arises for subject reduction, typically when a function expects two arguments on the same type: if this type is a union $U_1 \cup U_2$, the function might require its arguments to be both of type U_1 or both of type U_2, but this information can be "lost" during the reduction. We give an example derived from [BDCd95] [2].

Example 2.3.1 (Typing lost by reduction) . Let

$$\Gamma = x : (U_1 \to U_1 \to T) \cap (U_2 \to U_2 \to T) \ , \ y : U_1 \cup U_2 \ .$$

Then $\Gamma \vdash \lambda z.xzz : (U_1 \cup U_2) \to T$:

[2] For more complex union type system the counter-example to subject reduction is slightly more complex, but the idea remain the same.

$$\dfrac{\dfrac{\dfrac{\Gamma, z : U_1 \vdash x : U_1 \to U_1 \to T}{\Gamma, z : U_1 \vdash xzz : T} \quad \dfrac{\Gamma, z : U_2 \vdash x : U_2 \to U_2 \to T}{\Gamma, z : U_2 \vdash xzz : T}}{\Gamma, z : U_1 \cup U_2 \vdash xzz : T}\,\textbf{Union}}{\Gamma \vdash \lambda z.xzz : (U_1 \cup U_2) \to T}$$

Also it is possible to type $\Gamma \vdash (\lambda z.xzz)y : T$, but $(\lambda z.xzz)y$ β-reduces on xyy, that is not typable under context Γ: in order to apply x to argument y we cannot eliminate the intersection in its type, as we do not know whether y has type U_1 or U_2. We can neither commute the arrow and the intersection: we would obtain

$$\dfrac{\dfrac{(U_1 \to U_1 \to T) \cap (U_2 \to U_2 \to T) \preccurlyeq (U_1 \cup U_2) \to \big((U_1 \to T) \cup (U_2 \to T)\big)}{\Gamma \vdash x : (U_1 \cup U_2) \to \big((U_1 \to T) \cup (U_2 \to T)\big)}}{\Gamma \vdash xy : (U_1 \to T) \cup (U_2 \to T)}\,\textbf{\to elim}$$

and then it is not possible to conclude (at this stage, we actually lost information: even if we knew whether type of y is U_1 or U_2, we could not conclude that xyy has type T knowing that xy has type $(U_1 \to T) \cup (U_2 \to T)$). ⌟

Also there is no simple way to build a type system with union types that enjoys subject reduction, and pattern matching requires union types. Yet, [BDCd95] prove a kind of "big step subject reduction" for typed lambda calculus with union: If a term t has type T, and $t \to_\beta u$, then there is a term s of type T such that $u \to_\beta^* s$. In next chapter, we will prove a similar result for data structures of typed $\lambda_{\mathscr{C}}$-calculus using a reducibility model.

2.3.2 About strong normalisation and match failure

Typed lambda calculus with constructors supports some non-terminating reductions, and also match failures can occur. This is due to one of the administrative[3] rules: the composition of case-bindings. We first present a counter-example to strong normalisation, before taking out rule CaseCase from the calculus.

The problem of case-composition

Typed $\lambda_{\mathscr{C}}$-calculus does not prevent match failure. Indeed, rule CaseCase can create sub-terms whose typing is not checked in the "dead branches" of a case-binding. For instance, if $\phi = \{\mathsf{d} \mapsto \mathsf{d}'\}$ and $\theta = \{\mathsf{c} \mapsto \mathsf{d}\,; \mathsf{c}' \mapsto \mathsf{c}'\}$, then

$$\vdash \phi : \mathbf{d} \to \mathbf{d}' \qquad \text{and} \qquad \vdash \theta : \mathbf{c} \to \mathbf{d}.$$

So we can derive $\vdash \{\!|\theta|\!\} \cdot \mathsf{c} : \mathbf{d}$ and then $\vdash \{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot \mathsf{c} : \mathbf{d}'$. This makes sense because we can reduce $\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot \mathsf{c} \to^* \mathsf{d}'$ using twice the rule CaseCons. In θ, $\mathsf{c}' \mapsto \mathsf{c}'$ is a dead branch and is forgotten by the typing (once we know that c' itself is typable). However, we can also apply the rule CaseCase and get

[3] Rule CaseCase is not necessary in a reduction to reach a value, as it is formally expressed in Sec. 3.2.3.

$\{\!|\phi \circ \theta|\!\} \cdot \mathsf{c}$. Hence, the second branch of the case-binding is $\mathsf{c}' \mapsto \{\!|\phi|\!\} \cdot \mathsf{c}'$, which raises a match failure and is definitely not typable.

The point is that, while typing a case binding, a choice can implicitly be made concerning the branches that will be taken in consideration (if we had chosen type $\mathbf{c}' \to \mathbf{c}'$ for θ, we would not have been able to type $\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot \mathsf{c}'$, that reduces on the same match-failing term $\{\!|\phi|\!\} \cdot \mathsf{c}'$). Yet rule CaseCase can create redexes in branches that have been dropped by the typing.

For the same reason, rule CaseCase together with rule AppLam makes some typable terms non-terminating: let $\phi = \{\mathsf{d} \mapsto \delta\}$ and $\theta = \{\mathsf{c} \mapsto \mathsf{d}\ ; \mathsf{c}' \mapsto \mathsf{d}\delta\}$, where $\delta = \lambda x.xx$. Then we can derive

$$\dfrac{\Gamma \vdash \phi : \mathbf{d} \to \Delta \qquad \dfrac{\dfrac{\Gamma \vdash \mathsf{d} : \mathbf{d} \qquad \Gamma \vdash \mathsf{d}\delta : \mathbf{d}\Delta}{\Gamma \vdash \theta : \mathbf{c} \to \mathbf{d}} \qquad \Gamma \vdash x : \mathbf{c}}{\Gamma \vdash \{\!|\theta|\!\} \cdot x : \mathbf{d}}}{\Gamma \vdash \{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot x : \Delta}$$

with $\Gamma = x : \mathbf{c}$, and $\Delta = (\forall X.X \to X) \to (\forall X.X \to X)$. It appears that $\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot x$ is in normal form without rule CaseCase, but with it we can reduce

$$\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot x \to \{\!|\phi \circ \theta|\!\} \cdot x = \left\{\!\!\left|\begin{array}{l} \mathsf{c} \mapsto \{\!|\phi|\!\} \cdot \mathsf{d} \\ \mathsf{c}' \mapsto \{\!|\phi|\!\} \cdot \mathsf{d}\delta \end{array}\right|\!\!\right\} \cdot x \to^* \left\{\!\!\left|\begin{array}{l} \mathsf{c} \mapsto \delta \\ \mathsf{c}' \mapsto \delta\delta \end{array}\right|\!\!\right\} \cdot x$$

Hence $\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot x$ is not normalising because of the sub-term $\delta\delta$.

Restricted lambda calculus with constructors

The rule CaseCase was introduced in the lambda calculus with constructors in order to satisfy the separation property [AMR09, Theo. 2] —and the same for rule LamApp, the usual η-reduction. Yet it is unessential for computing in the lambda calculus with constructors (*cf.* discussion of Sec. 3.2.3). Since it leads to some match failure or non-terminating reduction in the typed calculus, we take it away and we consider $\lambda^{-}_{\mathscr{C}}$, the lambda calculus with constructors restricted to eight reduction rules: AppLam, LamApp, CaseCons, CaseApp, CaseLam, AppDai, LamDai and CaseDai. This calculus is known to be confluent [AMR09, Theo. 1].

In next chapter, we provide a denotational model for typed $\lambda^{-}_{\mathscr{C}}$-calculus, that ensures its strong normalisation and guaranties the absence of match failure.

Conclusion and future work

The lambda calculus with constructors presents some reduction rules that do not match with usual typing intuitions, in particular the commutation rule between application and case construct. To cope with this peculiarity, we have defined a complex type system, with a sub-typing rule transforming a data-type into an arrow type. This type system is polymorphic, and includes intersection and union

types. There is also a new construction: the type application. In the next chapter, we show that it is a correct type system *w.r.t.* strong normalisation and prevention of match failure for the restricted $\lambda_{\mathscr{C}}$-calculus (without rule CaseCase).

A natural question would be the converse: is any perfectly normalising term of $\lambda^{-}_{\mathscr{C}}$-calculus typable in some context? In the pure lambda calculus, every strongly normalising term is typable in the type system with arrow and intersection [Gal98, Kri93]. In $\lambda^{-}_{\mathscr{C}}$-calculus the answer would be negative without polymorphism. Consider for instance term $t = \lambda x.\{\!|\mathsf{c} \mapsto \mathsf{d}|\!\}\cdot\{\!|\mathsf{c} \mapsto \mathsf{d}|\!\}\cdot x$. It is a defined normal form (since we removed rule CaseCase). Without the rule **Cb**, the only type we could give to $u = \{\!|\mathsf{c} \mapsto \mathsf{d}|\!\}\cdot x$ would be $\mathbf{d}$ (with x of type $\mathbf{c}$). Then $\{\!|\mathsf{c} \mapsto \mathsf{d}|\!\}\cdot u$ would not be typable. However, with the rule **Cb** we can type the case bindings $\{\mathsf{c} \mapsto \mathsf{d}\}$ with type $\forall\alpha.\alpha \to \forall X.X$, and then derive $\vdash t : \forall\alpha.\alpha \to \forall X.X$ (since $\forall X.X \preccurlyeq \forall\alpha.\alpha$).

Also we have good hope that the type system described in this chapter is able to type any perfectly normalising $\lambda^{-}_{\mathscr{C}}$-term. A first attempt to show this could be to assign type $\forall X.X$ to every variable, and then follow the proof method of [Gal98]: first show that every defined normal form (they are on the form $\lambda\vec{x}.\mathsf{c}\,\vec{u}$ or $\lambda\vec{x}.\{\!|\theta_1|\!\}\cdot\cdots\cdot\{\!|\theta_k|\!\}\cdot y\,\vec{u}$ with possibly empty vectors and $k \geq 0$) has a type. Then proceed by induction on derivation of a perfectly normalising term.

An other issue that we let open is the decidability of (a sub-system of) the type system. Two questions can be asked:

1. Type Inference: Given a term t, is there a context Γ and a type T such that $\Gamma \vdash t : T$?

2. Type checking: Is a judgement $\Gamma \vdash t : T$ derivable?

Notice that, in general, the decidability of type checking entails the one of type inference. Indeed, if we call *contexted typability* the question of knowing, given a term t and a context Γ, if t can be typed in context Γ, then

- Type inference is decidable if contexted typability is. To know whether t can be typed in some context, ask if $\lambda x_1 \ldots x_n.t$ is typable in the empty context (where $\{x_1, \ldots, x_n\} = \mathrm{fv}(t)$).

- Contexted typability is decidable if type checking is. To know if t is typable in context Γ, ask if $\Gamma, z : Z \vdash (\lambda xy.y)\ t\ z : Z$ is derivable.

If, as we conjectured previously, our type system is complete for perfectly normalising terms, type inference is certainly undecidable. Indeed, within the sub-family of pure lambda terms, the perfect normalisation is equivalent to the strong normalisation, that is known to be undecidable (a nice proof can be found in [Urz03]). It is not surprising as both type inference and type checking are undecidable in Curry-style *System F* [Wel94]. Also the simply typed lambda calculus with intersection types is complete for strong normalising terms [vB92, Ghi96].

Yet the question can be asked for the type system without polymorphism and without intersection type. This would require an other rule for typing case-bindings (the one given in this chapter is too weak without intersections), such as for instance

$$\frac{\Gamma \vdash u_i : \vec{U_i} \to T_i \quad (1 \leq i \leq n)}{\Gamma \vdash \{\mathsf{c}_i \mapsto u_i / 1 \leq i \leq n\} : (\bigcup_{i=1}^{n} \mathbf{c}_i \vec{U_i}) \to (\bigcup_{i=1}^{n} T_i)}$$

However even with such a rule, an inference algorithm (if it exists) would probably need a backtracking mechanism, since the arrow types (of the u_i's) can have several (but a finite number of) vectorial notations (Rem. 2.2.1).

Altthough this question is essential in the prospect of implementing the lambda calculus with constructors, it is far from being trivial.

Chapter 3

A Reducibility Model

In this chapter, we shall prove the strong normalisation theorem for the restricted typed $\lambda_{\mathscr{C}}$-calculus using Girard's technique of reducibility candidates [GLT89, Chap.14]. The main idea is to interpret every type T by a particular set $|T|$ of strongly normalising terms called *reducibility candidates*. We then prove that this interpretation is *sound w.r.t.* typing, *i.e.* every term of type T is in its interpretation —and thereby is strongly normalising.

Reducibility candidates allow a finer analysis of terms than types do. Some properties (such as strong normalisation) we want to prove for typable terms can actually be set in the definition of reducibility candidates. Their definition must also include some closure properties, that ensure soundness.

The proof by reducibility candidates of this chapter presents three main novelties. The first one is to focus not only on strong normalisation, but also on well-definition. This implies to restrict candidates to *closed terms* (what would it mean for $\{\!|\theta|\!\} \cdot x$ to be defined?). This is why the Daimon shall play an important role. Next we need to adapt the usual definition of candidates to the application types. Therefore we introduce a closure operator on candidates. It shall also enable a fine-grained analysis of data-structures and data-types (in particular we introduce the notion of *data-candidates*). Finally, we ensure that candidates are closed under union, using Riba's techniques [Rib07b].

For the reasons explained in Sec. 2.3.2, we consider here a restriction of the calculus, without rule CaseCase, called the $\lambda^{-}_{\mathscr{C}}$-calculus. Also, throughout this chapter,

$$\rightarrow \textit{ denotes a reduction step that does not use } \textsc{CaseCase} \ .$$

3.1 Reducibility candidates

Reducibility candidates are defined by a predicate on terms: they must satisfy some conditions that exclude "bad" terms, but also some closure conditions that will entail soundness for the reducibility model. As usual, those are conditions of closure under reduction, or under some expansion.

Notations: A term is said to be *neutral* if it is not a value. We write $\mathscr{N}_D$ the set of closed defined and neutral terms, and PN_o the set of closed and perfectly normalising terms. The set of closed terms is denoted by Λ_o. Given a term t, $Red(t)$ is the set of its reducts (for the rules of $\lambda^{-}_{\mathscr{C}}$) in one step, and $Red_*(t)$ denotes the set of its reducts in any number of steps (including zero).

Definition 3.1.1 (*Reducibility candidates*)

A set S of closed terms is a *reducibility candidate* when it satisfies:

(CR_1) Perfect normalisation: $S \subseteq PN_o$.

(CR_2) Stability by reduction: $t \in S \Rightarrow Red(t) \subseteq S$.

(CR_3) Stability by neutral expansion:
if $t \in \mathscr{N}_D$, then $Red(t) \subseteq S \;\Rightarrow\; t \in S$.

(CR_4) Stability by case-commutation:
if $t \to_{\textsc{CaseApp}} t'$ or $t \to_{\textsc{CaseLam}} t'$, and $t' \in S$, then $t \in S$.

We call (CR) the conjunction of (CR_1), (CR_2), (CR_3) and (CR_4).

The first three conditions correspond to the ones of the original definition of Girard, taking care to rule out undefined terms. The last one is necessary for the soundness result, because of the "ill behaviour" of rules CaseApp and CaseLam *w.r.t.* typing (*cf.* Rem. 2.1.2). In fact, (CR_4) means that reducibility candidates do not separate terms that are CaseApp or CaseLam equivalent. Once we will have proved that this leads to a relevant notion of candidates (that is, it is compatible with the property of perfect normalisation), it will informally allow us to consider terms up to CaseApp and Case Lam. This requires to treat these two rules with a special attention (which is the purpose of next section).

Taking up type terminology, we call *data-candidate* a reducibility candidate whose values all are data-structures. In other words, a reducibility candidate is a data-candidate if it contains no abstraction $\lambda x.t$. We write $\mathcal{CR}$ the class of reducibility candidates, and $\mathcal{DC}$ its sub-class of data candidates.

Reducibility candidates are usually not restricted to closed terms. On the contrary, free variables are needed in reducibility candidates as *hereditarily neutral* terms, *i.e.* as terms with no values in their reducts. Indeed, hereditarily neutral terms insure that reducibility candidates are not empty (we detail this in Rem. 3.1.4). Since the $\lambda^{-}_{\mathscr{C}}$-calculus is provided with the Daimon, which is also hereditarily neutral, open terms are useless in reducibility candidates. Moreover, keeping only closed terms circumscribes candidates to terms whose meaning is entirely known.

Condition (CR_2) can be expressed in a slightly different way, that we may use later. We call (CR'_2) this equivalent condition.

Lemma 3.1.1 (Condition (CR'_2))**.** *For any set of terms S, the following conditions are equivalent:*

(CR_2) *For every term t in S, $Red(t) \subseteq S$*

(CR'_2) *For every term t in S, $Red_*(t) \subseteq S$*

PROOF : *Condition (CR'_2) obviously implies (CR_2). Now assume S satisfies (CR_2). By induction on n it is trivial to show that $t \in S$ and $t \to^n t'$ imply $t' \in S$ for all $n \geq 0$. So $t \in S$ implies $Red_*(t) \subseteq S$.* □

3.1.1 Case normal form

This section gives some technical details about case-commutation. We first show that every term has a normal form for case-commutation, and we give an explicit definition of it. then we prove that case-commutation equivalence is compatible with perfect normalisation, which provides a new definition of reducibility candidates.

The case-commutation rules are CASEAPP and CASELAM. Remember that we note $\rightharpoonup$ a reduction step by one of them, and $\to_{\mathcal{B}}$ a reduction step by one of the other rule of $\lambda^{-}_{\mathscr{C}}$ (APPLAM, LAMAPP, CASECONS, APPDAI, LAMDAI, or CASEDAI).

Every term is strongly normalising for $\rightharpoonup$. Indeed, reducing a term t with a case-commutation rule decreases its *structural measure* $s(t)$ (Def. 2.1.2). Moreover, $\rightharpoonup$ is confluent (Sec. 2.1.5). Thus every term t has a normal form for $\rightharpoonup$, that we call its *case normal form* and that we write $\downarrow t$. It is characterised by the following equations:

$$
\begin{array}{rcl@{\qquad}rcl}
\downarrow x &=& x & \downarrow\{\!|\theta|\!\}\cdot x &=& \{\!|\downarrow\theta|\!\}\cdot x \\
\downarrow \mathsf{c} &=& \mathsf{c} & \downarrow\{\!|\theta|\!\}\cdot \mathsf{c} &=& \{\!|\downarrow\theta|\!\}\cdot \mathsf{c} \\
\downarrow \maltese &=& \maltese & \downarrow\{\!|\theta|\!\}\cdot \maltese &=& \{\!|\downarrow\theta|\!\}\cdot \maltese \\
\downarrow \lambda x.t &=& \lambda x.\downarrow t & \downarrow\{\!|\theta|\!\}\cdot \lambda x.t &=& \lambda x.\downarrow(\{\!|\theta|\!\}\cdot t) \\
\downarrow (tu) &=& \downarrow t \downarrow u & \downarrow\{\!|\theta|\!\}\cdot (tu) &=& \downarrow(\{\!|\theta|\!\}\cdot t)\downarrow u \\
\downarrow\{\mathsf{c}_i \mapsto u_i \,/\, 1\leq i\leq n\} &=& \{\mathsf{c}_i \mapsto\downarrow u_i \,/\, 1\leq i\leq n\} & \downarrow\big(\{\!|\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t\big) &=& \downarrow(\{\!|\theta|\!\}\cdot\downarrow\{\!|\phi|\!\}\cdot t) \\
\multicolumn{3}{r}{\text{if } \downarrow\{\!|\phi|\!\}\cdot t = \{\!|\phi|\!\}\cdot t, \text{ then}} & \downarrow\big(\{\!|\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t\big) &=& \{\!|\downarrow\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t
\end{array}
$$

To deal with perfect normalisation, we can consider terms up to case-commutation. Indeed, both well-definition and strong normalisation are preserved by case-commutation equivalence. That is what Cor. 3.1.4 expresses.

Lemma 3.1.2. *If $\downarrow t$ is defined, so is t.*

PROOF : *Using the characterisation of $\downarrow t$, it is easy to check by induction on t that if t is undefined, then $\downarrow t$ also is.* □

Remark 3.1.1 . Since $t \in \mathcal{V}$ implies $\downarrow t \in \mathcal{V}$, it follows that

$$\downarrow t \in \mathscr{N}_D \implies t \in \mathscr{N}_D$$

Lemma 3.1.3. *For every terms t, t', if $t \to_{\mathcal{B}} t'$ then $\downarrow t \to^+ \downarrow t'$*

PROOF : *By structural induction on t.*

1. *If t is a variable, the Daimon or a constructor, then t is not reducible.*
2. *If $t = \lambda x.t_0$, then necessarily $t' = \lambda x.t'_0$ with $t_0 \to_{\mathcal{B}} t'_0$ and we conclude by induction.*
3. *If $t = t_1 t_2$, three different cases can occur:*
   (a) *$t' = t_1 t'_2$ or $t'_1 t_2$ with $t_i \to_{\mathcal{B}} t'_i$. Hence we conclude by induction*
   (b) *$t_1 = \maltese$ and $t' = \maltese$. In that case $\downarrow t = (\maltese \downarrow t_2)$ reduces to $\maltese = \downarrow t'$.*
   (c) *$t_1 = \lambda x.t_0$ and $t' = t_0[x := t_2]$. Then $\downarrow t = (\lambda x.\ \downarrow t_0)\downarrow t_2$, and it reduces to $(\downarrow t_0)[x :=\downarrow t_2]$, that has case normal form (and therefore reduces in 0 or more steps on) $\downarrow(t_0[x := t_2])$.*
4. *If $t = \{\!|\theta|\!\} \cdot t_0$, either $t' = \{\!|\theta'|\!\} \cdot t_0$ or $\{\!|\theta|\!\} \cdot t'_0$ with $\theta \to_{\mathcal{B}} \theta'$ or $t_0 \to_{\mathcal{B}} t'_0$ and we conclude by induction, or $t' = u$ with $t_0 = \mathsf{c}$ and $\mathsf{c} \mapsto u \in \theta$, or $t' = \maltese$ and $t_0 = \maltese$. In both last cases, $\downarrow t = \{\!|\downarrow\theta|\!\} \cdot t_0 \to \downarrow t'$.* □

Corollary 3.1.4. *If $\downarrow t \in PN_o$, then $t \in PN_o$.*

Proof : *First $u \in Red_*(t)$ implies $\downarrow u \in Red_*(\downarrow t)$ by Lem. 3.1.3. Thus Lem. 3.1.2 entails that all reducts of t are defined as soon as all reducts of $\downarrow t$ are. That is, t is hereditarily defined if $\downarrow t$ is.*
Now assume there is an infinite reduction $t = t_0 \to t_1 \to t_2 \ldots$ Since $\rightharpoonup$ is strongly normalising, this reduction chain contains an infinity of $\to_{\mathcal{B}}$ reduction steps: $t = t_0 \rightharpoonup^* t_{i_1} \to_{\mathcal{B}} t_{j_1} \rightharpoonup^* t_{i_2} \to_{\mathcal{B}} t_{j_2} \ldots$
For every k, $\downarrow t_{j_k} = \downarrow t_{i_{k+1}}$ and $\downarrow t_{i_k} \to^+ \downarrow t_{j_k}$ by Lem. 3.1.3. Hence there is an infinite reduction

$$\downarrow t = \downarrow t_{i_1} \to^+ \downarrow t_{j_1} = \downarrow t_{i_2} \to^+ \downarrow t_{j_2} = \downarrow t_{i_3} \to^+ \downarrow t_{j_3} \ldots$$

This is absurd if $\downarrow t$ is strongly normalising. So finally if $\downarrow t$ is perfectly normalising then t strongly normalises too. □

This corollary allows us to formulate differently the last condition in the definition of reducibility candidates. We call (CR'_4) this alternative condition.

Lemma 3.1.5 (Condition (CR'_4))**.** *We say that a set of terms S satisfies (CR'_4) when*

$$\text{for every term } t, \quad \downarrow t \in S \text{ implies } t \in S. \tag{CR'_4}$$

Then, for any set of terms S, $(\mathrm{CR}_2) \wedge (\mathrm{CR}_4)$ is equivalent to $(\mathrm{CR}'_2) \wedge (\mathrm{CR}'_4)$.

Proof : *First remember that (CR_2) is equivalent to (CR'_2) (Lem. 3.1.1). Now assume S satisfies (CR_4). If t is a term such that $\downarrow t \in S$, we can see by induction on the reduction $t \rightharpoonup^* \downarrow t$ that $t \in S$. So (CR'_4) holds. Conversely, if S satisfies (CR'_2) and (CR'_4), then for any $t' \in S$ and any $t \rightharpoonup t'$, we have $\downarrow t = \downarrow t'$ is in S by (CR'_2) (since $t' \to^* \downarrow t'$), thus $t \in S$ by (CR'_4).* □

In the following, to characterise a reducibility candidate, we may use either $(\mathrm{CR}_1) \wedge (\mathrm{CR}_2) \wedge (\mathrm{CR}_3) \wedge (\mathrm{CR}_4)$, or $(\mathrm{CR}_1) \wedge (\mathrm{CR}'_2) \wedge (\mathrm{CR}_3) \wedge (\mathrm{CR}'_4)$ depending on what is more convenient.

Remark 3.1.2 . The set PN_o is a reducibility candidate: it obviously satisfies (CR_1) and (CR_2), and also (CR_3) since a defined closed term whose all reducts are in PN_o is in PN_o itself. Last, Cor. 3.1.4 entails that PN_o satisfies (CR'_4).

3.1.2 Closure property

In this section, we give a way to construct a reducibility candidate from a set of perfectly normalising terms. A *non-expanded candidate* is a set of terms that satisfies (CR_1) and (CR_2). Sets that satisfy (CR_4) in addition (or equivalently (CR'_4)) are called *pre-candidates* of reducibility. We write $\mathcal{PCR}$ for the family of pre-candidates. For instance $\{\mathsf{c}\}$ is a pre-candidate for any constructor c. We will see that such pre-candidates can be closed by (CR_3) to obtain a reducibility candidate.

Definition 3.1.2 (*Closure*)

For $X \subseteq \Lambda_o$, we note $\overline{X}$ its closure by (CR_3). It is defined inductively by

$$\frac{t \in X}{t \in \overline{X}} \qquad \frac{t \in \mathscr{N}_D \quad Red(t) \subseteq \overline{X}}{t \in \overline{X}}$$

Lemma 3.1.6 (Closure of a pre-candidate)**.** *If $P \in \mathcal{PCR}$, then $\overline{P}$ is the smallest reducibility candidate containing P.*

PROOF : *$\overline{P}$ satisfies (CR_3) by definition. Using the inductive definition, it is immediate to check that it satisfies (CR_1) and (CR_2). Now we prove that it satisfies (CR'_4). Let $t \in \Lambda_o$ such that $\downarrow t \in \overline{P}$, and show that $t \in \overline{P}$ by induction on its derivation.*

1. *If $\downarrow t \in P$ then $t \in P$ since $P \in \mathcal{PCR}$ and thus satisfies (CR'_4).*
2. *Else $\downarrow t \in \mathscr{N}_D$ and $Red(\downarrow t) \subseteq \overline{P}$. In that case, t also is in $\mathscr{N}_D$ (Rem. 3.1.1) and for all $u \in Red(t)$, $\downarrow u \in Red_*(\downarrow t)$ (Lem. 3.1.3). Moreover, $Red_*(\downarrow t) \subseteq \overline{P}$ by (CR'_2), thus $\downarrow u \in \overline{P}$. By induction hypothesis, it implies that $u \in \overline{P}$. Hence $Red(t) \subseteq \overline{P}$, so $t \in \overline{P}$ for being neutral.*

Finally $\overline{P}$ is a reducibility candidate. Moreover, if a reducibility candidate $\mathcal{A}$ contains P, it also contains $\overline{P}$ by (CR_3). So $\overline{P}$ is the smallest candidate containing P. □

In the previous lemma it would not be sufficient to assume that P is a non-expanded candidate, to conclude $\overline{P} \in \mathcal{CR}$, as shown by the following example.

Example 3.1.3 . Let $t = \lambda y.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot y$ and $u = \{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda y.y$. Then $u \rightharpoonup t$. The set $S = \{\lambda x.t\}$ satisfies (CR_1) and (CR_2) but $\overline{S}$ does not satisfy (CR_4) since $\lambda x.u \notin \overline{S}$. So $\overline{S}$ is not a reducibility candidate.

We now characterise precisely when a non-expanded candidate can be closed to obtain a reducibility candidate.

Lemma 3.1.7 (Closure of a non-expanded candidate)**.** *Let S be a non-expanded candidate. Then $\overline{S}$ is a reducibility candidate if and only if, for any $t, t' \in \Lambda_o$,*

$$\left.\begin{array}{l} t \rightharpoonup t' \\ t' \in S \end{array}\right\} \implies t \in \overline{S}$$

PROOF : *The implication ("only if" side) is trivial by* (CR_4)*. We prove the converse. Assume that for any $t, t' \in \Lambda_o$, $t \rightharpoonup t'$ with $t' \in S$ implies $t \in \overline{S}$. By definition $\overline{S}$ satisfies* (CR_3)*. The closure operator $\overline{\cdot}$ preserves* (CR_1) *and* (CR_2)*, so these two properties also hold in $\overline{S}$. Now, we need to prove* (CR'_4)*. Let $\downarrow t \in \overline{S}$. By Cor. 3.1.4, $\downarrow t \in PN_o$ implies $t \in PN_o$. We prove by induction on its reduction that $t \in \overline{S}$. If $t = \downarrow t$ it is clear; else let t' such that $t \rightharpoonup t' \rightharpoonup^* \downarrow t$. By induction hypothesis, $t' \in \overline{S}$.*

1. *If $t' \in S$ then by hypothesis $t \in \overline{S}$.*
2. *Otherwise $t' \in \mathscr{N}_D$ and $Red(t') \in \overline{S}$ (by definition of the closure operator). Hence t also is in $\mathscr{N}_D$ (same as Rem. 3.1.1). Moreover given $u \in Red(t)$, $\downarrow t \to^* \downarrow u$ by Lem. 3.1.3. So $\downarrow u \in \overline{S}$ by* (CR_2)*, and $u \in \overline{S}$ by induction hypothesis. Thus $Red(t) \subseteq \overline{S}$ and $t \in \overline{S}$.*

Hence $\overline{S}$ satisfies also (CR'_4)*, it is then a reducibility candidate.* □

Remark 3.1.4 . Stability under (CR_3) also entails that all reducibility candidates are infinite: first they are non-empty since they all contain the Daimon, as neutral and irreducible term. Moreover, if $\mathcal{A} \in \mathcal{CR}$ contains a term t, it also contains $\{\!|\mathsf{c} \mapsto t|\!\} \cdot \mathsf{c}.$ as a neutral term whose all reducts (by induction on the reduction of t) are in $\mathcal{A}$. So we can construct an infinite family of (different) terms in $\mathcal{A}$. Even the smallest reducibility candidate, the closure of the empty set $\overline{\emptyset}$ is infinite. In fact it is often more relevant to focus on the *values* of a reducibility candidates, as it is formalised in next section.

Remember that a reducibility candidate whose all values are data-structures is called a *data-candidate*. The class of data-candidates, $\mathcal{DC}$, will be helpful to interpret data-types.

Remark 3.1.5 . Since the closure by (CR_3) only adds neutral terms, if P is a pre-candidate whose all values are data-structures, then $\overline{P} \in \mathcal{DC}$. In particular $\overline{\{\mathbf{c}\}}$ is a data-candidate for any constructor c.

3.1.3 Reducibility candidates and values

In this section we show that reducibility candidates are completely defined by their values, and also that they satisfy a property called the *principal reduct* property, that will be needed in the following section.

First notice that Prop. 2.1.1 remains valid in $\lambda^{-}_{\mathscr{C}}$-calculus.

Proposition 3.1.8. *Every defined closed term that is irreducible for $\lambda^{-}_{\mathscr{C}}$-rules is either the Daimon or a value.*

PROOF : *Same proof as the one of Prop. 2.1.1, as it does not use rule* CASECASE. □

A reducibility candidate is stable under reduction and under expansion for neutral terms. As a consequence, it is entirely determined by its values. We call *values of a term* t (or of a set of terms S), and we write $\mathcal{V}(t)$ (*resp.* $\mathcal{V}(S)$), the set of values to which t (*resp.* a term of S) reduces:

$$\mathcal{V}(t) = Red_*(t) \cap \mathcal{V}$$

Remark that, given $\mathcal{A}$ a reducibility candidate, $\mathcal{V}(A)$ is *a priori* not a pre-candidate, even not a non-expanded candidate. Indeed, $\mathcal{V}$ is not necessarily closed by reduction because of rules LamApp and LamDai. It is neither closed by (CR_4)because of rule CaseLam.

Example 3.1.6 . Consider the reducibility candidate $\overline{S}$, with

$$S = \{\ \lambda x.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot x \quad ; \quad \{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda x.x\ \}\ .$$

(It is actually a reducibility candidate by Lem. 3.1.6 since $S \in \mathcal{PCR}$). Then $\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda x.x \quad \rightharpoonup \quad \lambda x.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot x$, whereas $\lambda x.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot x$ is in $\mathcal{V}(\overline{S})$ and $\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda x.x$ is not. So $\mathcal{V}(\overline{S})$ is not closed under (CR_4). ⌟

Also it is generally not possible to use the closure operator on a set of values $\mathcal{V}(S)$ to construct a reducibility candidate. However, the values of a reducibility candidate are, to some extent, sufficient to define it (Cor. 3.1.10).

Lemma 3.1.9. *If $t \in PN_o$ and $\mathcal{A} \in \mathcal{CR}$, then* $\quad t \in \mathcal{A} \ \Leftrightarrow\ \mathcal{V}(t) \subseteq \mathcal{A}$.
In particular, if $\mathcal{A} \in \mathcal{CR}$, then $\mathcal{A} = \overline{\mathcal{V}(\mathcal{A})}$.

Proof : *The implication is obvious using* (CR'_2). *We prove the converse by induction on the reduction of t (that is well-founded for strongly normalising terms). Assume $\mathcal{V}(t) \subseteq \mathcal{A}$ and prove that $t \in \mathcal{A}$. If t is a value it is clear since $t \in \mathcal{V}(t)$. Otherwise $t \in \mathscr{N}_D$, and for all u in $Red(t)$, $u \in \mathcal{A}$ by induction hypothesis (since $\mathcal{V}(u) \subseteq \mathcal{V}(t) \subseteq \mathcal{A}$). So $t \in \mathcal{A}$ by* (CR_3). □

Corollary 3.1.10. *Let $\mathcal{A}, \mathcal{B} \in \mathcal{CR}$. Then $\mathcal{V}(\mathcal{A}) = \mathcal{V}(\mathcal{B})$ iff $\mathcal{A} = \mathcal{B}$.*

Proof : *We show the implication, the converse is obviously true. Let $\mathcal{A}, \mathcal{B} \subseteq \mathcal{CR}$, such that $\mathcal{V}(\mathcal{A}) = \mathcal{V}(\mathcal{B})$. By Lem. 3.1.9,*

$$\begin{array}{rcl} t \in \mathcal{A} & \Leftrightarrow & \mathcal{V}(t) \subseteq \mathcal{A} \\ & \Leftrightarrow & \mathcal{V}(t) \subseteq \mathcal{V}(\mathcal{A}) \\ & \Leftrightarrow & \mathcal{V}(t) \subseteq \mathcal{V}(\mathcal{B}) \\ & \Leftrightarrow & \mathcal{V}(t) \subseteq \mathcal{B} \\ & \Leftrightarrow & t \in \mathcal{B} \end{array}$$

□

The characterisation of a reducibility candidate by its values will be used in the next section to prove that the class $\mathcal{CR}$ is stable under union. For that, we also use a sufficient condition described in [Rib07b]: the *principal reduct* property.

Lemma 3.1.11 (Principal reduct property)**.** *Every reducible term $t \in \mathscr{N}_D$ has a reduct (in one step) $u \in \Lambda_o$ such that*

$$t \to^* v \ \wedge\ v \in \mathcal{V} \qquad \Rightarrow \qquad u \to^* v$$

A term u that satisfies such a property is called a principal reduct *of t.*

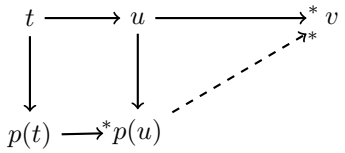
PROOF : *We inductively define, for every $t \in \mathscr{N}_D$ that can reduce on a value, the term $p(t)$:*

$$\begin{array}{rcl} p((\lambda x.t_0)t_1 \dots t_k) & = & t_0[x := t_1]\ t_2 \dots t_k \\ p((\{\!|\theta|\!\} \cdot t_0)t_1 \dots t_k) & = & p(\{\!|\theta|\!\} \cdot t_0)t_1 \dots t_k \\ p(\{\!|\theta|\!\} \cdot \mathsf{c}) & = & u \quad \text{if } \mathsf{c} \mapsto u \in \theta \\ p(\{\!|\theta|\!\} \cdot \lambda x.t_0) & = & \lambda x.\{\!|\theta|\!\} \cdot t_0 \\ p(\{\!|\theta|\!\} \cdot t_1 t_2) & = & (\{\!|\theta|\!\} \cdot t_1)t_2 \\ p(\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t_0) & = & \{\!|\theta|\!\} \cdot p(\{\!|\phi|\!\} \cdot t_0) \end{array}$$

By structural induction on t, it is immediate to check that

1. *$t \to p(t)$, and*
2. *if $t \to u$, either $u = p(t)$, or $p(t) \to^* p(u)$ (for the first case, remember that if $t_0 \to t_0'$ then $t_0[x := t_1] \to t_0'[x := t_1]$ by [AMR09, Lem. 9])*

This insures that $p(t)$ is a principal reduct of t: we show that $t \to^* v$ with $v \in \mathcal{V}$ implies $p(t) \to^* v$ by induction on the reduction $t \to^* v$. Since t is not a value, there is at least one step in the reduction. Let u be the first reduct. By induction hypothesis, there is a reduction $p(u) \to^* v$, and we can conclude since $p(t) \to^* p(u)$:



Hence for any $t \in \mathscr{N}_D$, and any $v \in \mathcal{V}$, if t reduces on v, then $p(t)$ also does. So $p(t)$ is a principal reduct of t. □

3.1.4 Candidates operators

In the next part, we define a model of the $\lambda^{-}_{\mathscr{C}}$-calculus, that interprets every type by a reducibility candidate. Therefore, we need to interpret basic types (Rem. 3.1.5 might give a hint), but also type operators in $\mathcal{CR}$. In this section, we define arrow, application, union and intersection for reducibility candidates.

Arrow. The arrow is defined in the usual way.

Definition 3.1.3

Given S and S' two sets of terms, $S \to S'$ is defined by

$$S \to S' \ = \ \{\, t \ / \ \forall u \in S,\ tu \in S' \,\}$$

Lemma 3.1.12. *If S is a non-expanded candidate (i.e. a set of terms satisfying* (CR$_1$) *and* (CR$_2$)*) that is non-empty, and $\mathcal{A} \in \mathcal{CR}$, then $S \to \mathcal{A} \in \mathcal{CR}$.*

PROOF : (CR$_1$) *Let $t \in S \to \mathcal{A}$. There exists $u \in S$, and $tu \in \mathcal{A} \subseteq PN_o$. So $t \in PN_o$.*

(CR$_2$) *Let $t \in S \to \mathcal{A}$ and $t' \in Red(t)$. For any $u \in S$, $tu \to t'u$. So $tu \in \mathcal{A}$ implies $t'u \in \mathcal{A}$ since $\mathcal{A}$ is closed under reduction. Hence $t' \in S \to \mathcal{A}$.*

(CR_3) *For any $t \in \mathscr{N}_D$ such that $Red(t) \subseteq S \to \mathcal{A}$, we prove that $u \in S$ implies $tu \in \mathcal{A}$ by induction on the reduction of u. Since $t \in \mathscr{N}_D$, tu is not a data-structure so $tu \in \mathscr{N}_D$. Furthermore t is not an abstraction so every reduct of tu is either $\maltese$ (if $t = \maltese$), or $t'u$ with $t' \in Red(t)$, or tu' with $u \to u'$. In any case it belongs to $\mathcal{A}$: $\maltese$ by (CR_3), $t'u$ because $t' \in S \to \mathcal{A}$, and tu' by induction hypothesis. So $tu \in \mathcal{A}$ by (CR_3), thus $t \in S \to \mathcal{A}$.*

(CR_4) *Let $t \rightharpoonup t'$ such that $t' \in S \to \mathcal{A}$. For any $u \in S$, $tu \rightharpoonup t'u$ and $t'u \in \mathcal{A}$. So $tu \in \mathcal{A}$ by (CR_4) in $\mathcal{A}$.*

Finally $S \to \mathcal{A}$ is a reducibility candidate. □

This lemma is stronger than the one we usually need:

$$\mathcal{A}, \mathcal{B} \in \mathcal{CR} \qquad \text{implies} \qquad \mathcal{A} \to \mathcal{B} \in \mathcal{CR} \tag{3.1}$$

Indeed, this proposition is entailed by Lem. 3.1.12 because every reducibility candidate is non-empty (they all contain the Daimon, *cf.* Rem. 3.1.4). However we relax the hypothesis in the lemma for a subsequent need.

Application. The application of two sets of terms is defined in the expected way: for S and S' sets of terms,

$$S\,S' \;=\; \{t\,u \;/\; t \in S \text{ and } u \in S'\}$$

There is no reason for $\mathcal{CR}$ to be closed under application. Indeed, none of (CR_1), (CR_2), (CR_3) and (CR_4) is preserved by application. However, the following lemma will be sufficient to establish the model.

Lemma 3.1.13. *If $\mathcal{D} \in \mathcal{DC}$ and $\mathcal{A} \in \mathcal{CR}$, then $\overline{\mathcal{DA}} \in \mathcal{DC}$.*

Proof : *First notice that $\overline{\mathcal{DA}} = \overline{\mathcal{DA} \cup \{\maltese\}}$ (since $\maltese$ is neutral with no reduct, it is in the closure of any set). We call S the set $\mathcal{DA} \cup \{\maltese\}$, and we will first prove that it is a non-expanded candidate. Then we will prove that $t' \in S$ and $t \rightharpoonup t'$ imply $t \in \overline{S}$. Also $\overline{S} \in \mathcal{CR}$ will result from Lemma 3.1.7.*

1. *Let $t \in S$. If t is the Daimon, it is perfectly normalising and it has no reduct. Otherwise, $t = t_1 t_2$ with $t_1 \in \mathcal{D}$ and $t_2 \in \mathcal{A}$. We show by induction on their reduction that $t \in PN_o$ and $Red(t) \subseteq S$. Term t_1 is not an abstraction since it is in a data candidate, so every reduct of t is either $\maltese$ (if $t_1 = \maltese$), or a term on the form $t_1' t_2$ or $t_1 t_2'$ with $t_i \to t_i'$. All this reducts are in S, and they are perfectly normalising (possibly by induction hypothesis). So $Red(t) \subseteq S$ and $t \in PN_o$. Hence S satisfies (CR_1) and (CR_2).*
2. *Let $t \rightharpoonup t'$ such that $t' \in S$. Then $t' = t_1 t_2$ with $t_1 \in \mathcal{D}$ and $t_2 \in \mathcal{A}$. Either $t = t_1' t_2$ or $t_1 t_2'$ with $t_i' \rightharpoonup t_i$ (in that case $t \in \mathcal{DA}$ since $\mathcal{D}$ and $\mathcal{A}$ are closed by expansion for $\rightharpoonup$), or $t = \{\!|\theta|\!\} \cdot (t_0 t_2)$ and $t_1 = \{\!|\theta|\!\} \cdot t_0$. In the last case, $t \in \mathscr{N}_D$: both $\{\!|\theta|\!\} \cdot t_0$ and t_2 are defined (they are in reducibility candidates) so $\{\!|\theta|\!\} \cdot (t_0 t_2)$ also is defined, and it is not a value. We show that all its reducts are in $\overline{S}$. Note that t_0 is not an abstraction (if $t_0 = \lambda x.t_0'$ then $t_1 \to \lambda x.\{\!|\theta|\!\} \cdot t_0' \notin \mathcal{D}$), so a reduct u of $\{\!|\theta|\!\} \cdot (t_0 t_2)$ may have three different forms:*
   (a) *$u = t'$. Hence $u \in S \subseteq \overline{S}$.*

(b) $u = \{\!|\theta|\!\} \cdot \Lightthorpe$ *(if* $t_0 = \Lightthorpe$*). In that case* $u \in \mathscr{N}_D$ *and all its reducts in any number of steps until* $\Lightthorpe$ *are in* $\mathscr{N}_D$*, so* u *is in* $\overline{S}$*.*

(c) $u = \{\!|\theta'|\!\} \cdot (t'_0 t'_2)$ *with* $\theta \to \theta'$ *and* $t_i = t'_i$*, or* $\theta = \theta'$ *and* $t_i \to t'_i$*.*
In that case, $u \rightharpoonup u' = (\{\!|\theta'|\!\} \cdot t'_0)t'_2$*, and* $t' \to u'$ *so* $u' \in S$ *by* (CR$_2$)*. Thus* $u \in \overline{S}$ *by induction hypothesis.*

Hence any reduct of t *is in* $\overline{S}$*, and thus* $t \in \overline{S}$ *by* (CR$_3$)*.*

By Lemma 3.1.7, $\overline{\mathcal{DA}} = \overline{S} \in \mathcal{CR}$*. What is more, all values of* $\overline{\mathcal{DA}}$ *are in* $\mathcal{DA}$*, thus they are applications, so they are data-structures. Finally,* $\overline{\mathcal{DA}} \in \mathcal{DC}$*.* □

Notice that we consider the closure of set application for a data-candidate and a candidate. In general, the closure of the application of two reducibility candidates would *not* form a reducibility candidate, as shown in the following example. This is intuitively due to the same reason why we do not consider general type application, but we restrict it to data-types: good properties (among which the perfect normalisation property) are insured to be preserved by applying a term t to u if t is not (and does not reduce on) an abstraction.

Example 3.1.7 . Consider the reducibility candidate $\mathcal{A} = \overline{\{I\}}$, where $I = \lambda x.x$ Then $II \in \overline{\mathcal{AA}}$, but $II \to I$ and $I \notin \overline{\mathcal{AA}}$. Thus $\overline{\mathcal{AA}}$ is not closed under (CR$_2$) and thereby is not a reducibility candidate.

Intersection. As usual[1], reducibility candidates are well-designed for set intersection.

Lemma 3.1.14. *Let* $(\mathcal{A}_i)_{i\in\mathcal{I}}$ *be a family of reducibility candidates, and* $(\mathcal{D}_i)_{i\in\mathcal{I}}$ *a family of data-candidates. Then,* $\bigcap_{i\in\mathcal{I}} \mathcal{A}_i \in \mathcal{CR}$ *and* $\bigcap_{i\in\mathcal{I}} \mathcal{D}_i \in \mathcal{DC}$*.*

Proof : *Each of the conditions* (CR$_1$)*,* (CR$_2$)*,* (CR$_3$) *and* (CR$_4$) *is preserved by intersection, so* $\bigcap_{i\in\mathcal{I}} \mathcal{A}_i$ *and* $\bigcap_{i\in\mathcal{I}} \mathcal{D}_i$ *are reducibility candidates. Moreover, the values of* $\bigcap_{i\in\mathcal{I}} \mathcal{D}_i$ *are values of data-candidates, hence they all are data-structures. Hence,* $\bigcap_{i\in\mathcal{I}} \mathcal{D}_i \in \mathcal{DC}$*.* □

Union. For a long time, union has been considered as problematic for Girard's reducibility candidates (see for instance [Wer94, Sec. 3.8.4]). The main difficulty is that condition (CR$_3$) is not preserved by union: if S and S' are two sets closed under (CR$_3$), and t is a normal term such that $Red(t) \subseteq S \cup S'$, it does not mean that $Red(t) \subseteq S$ or $Red(t) \subseteq S'$. That is why we cannot conclude *in general* that $t \in S \cup S'$.

However, [Rib07b] proposes a detailed analysis of reducibility candidates, and highlights a sufficient condition for them to be stable under union: the so-called *principal reduct property*. This condition is valid in our calculus (Lem. 3.1.11). Also we adapt the proof of Riba to our definition of reducibility candidates in order to get their stability under union.

[1] The different versions [GLT89, Tai67, Par93] of reducibility candidates aim to interpret polymorphic type system, and the second-order universal quantification is always interpreted by a generalised intersection.

Lemma 3.1.15. *For any family $(P_i)_{i\in I}$, of pre-candidates, $\overline{\bigcup P_i} \subseteq \bigcup \overline{P}_i$.*

PROOF : *By induction on $t \in \overline{\bigcup P_i}$, we show that $t \in \overline{P}_j$ for some $j \in I$.*

1. *If $t \in \bigcup P_i$, then there is $j \in I$ such that $t \in P_j$, hence $t \in \overline{P_j}$*
2. *If $t \in \mathscr{N}_D$ and $Red(t) \subseteq \overline{\bigcup P_i}$, let u be a principal reduct of t. Then $\mathcal{V}(t) = \mathcal{V}(u)$ (Lem. 3.1.11). Since $u \in Red(t)$, $u \in \overline{P_j}$ for some j by induction hypothesis. So $\mathcal{V}(u) \subseteq \overline{P_j}$ by (CR_2), and using Lem. 3.1.9 we get $t \in \overline{P_j}$.* □

Corollary 3.1.16. *Let $(\mathcal{A}_i)_{i\in\mathcal{I}}$ be a family of reducibility candidates, and $(\mathcal{D}_i)_{i\in\mathcal{I}}$ a family of data-candidates. Then, $\bigcup_{i\in\mathcal{I}} \mathcal{A}_i \in \mathcal{CR}$ and $\bigcup_{i\in\mathcal{I}} \mathcal{D}_i \in \mathcal{DC}$.*

PROOF : *All candidates $\mathcal{A}_i$ satisfy (CR_3), thus $\mathcal{A}_i = \overline{\mathcal{A}_i}$ for any i. By Lem. 3.1.15, it means that $\overline{\bigcup \mathcal{A}_i}$ is included in $\bigcup \mathcal{A}_i$. The converse inclusion also holds by definition, so $\bigcup \mathcal{A}_i = \overline{\bigcup \mathcal{A}_i}$. Moreover, $\bigcup \mathcal{A}_i$ is pre-candidate since (CR_1), (CR_2) and (CR_4) are preserved by union. Thus $\overline{\bigcup \mathcal{A}_i}$ is a reducibility candidate (by Lem. 3.1.6), and so is $\bigcup \mathcal{A}_i$. In the same way, $\bigcup \mathcal{D}_i$ is a reducibility candidate, and all its values are in some data-candidate $\mathcal{D}_j$, so $\bigcup \mathcal{D}_i \in \mathcal{DC}$.* □

Finally, we have provided family $\mathcal{CR}$ with operators arrow, union and intersection, and family $\mathcal{DC}$ with union and intersection. Thanks to the closure operator, we also have an easy way to construct a new data-candidate by applying a data-candidate to an other reducibility candidate. Also the interpretation of types by reducibility candidates will be quite straightforward.

3.2 Denotational model

In this part we associate to every type T a reducibility candidate that contains all the terms which are typable by T. Seeing typed terms as terms of a reducibility candidate or a data-candidate will then enable a finer analysis of their properties.

3.2.1 Types interpretation

To achieve the definition of type interpretation, we need to give the interpretation for type variables. For that, we use *valuations*, i.e. functions matching every data-type variable to a data-candidate, and every type variable to a reducibility candidate.

Given a valuation ρ, the *interpretation* of a type T in ρ, written $[T]_\rho$, is defined inductively in Fig. 3.1. We also associate to T (seen as a type for case bindings) and ρ the set of case bindings $[\![T]\!]_\rho$.

Lemma 3.2.1. *For every type T, if ρ is a valuation such that $\mathrm{dom}(\rho) \subseteq \mathrm{TV}(T)$, then $[T]_\rho \in \mathcal{CR}$. Moreover, if T is a data-type, then $[T]_\rho \in \mathcal{DC}$.*

PROOF : *By structural induction on the type T, using Rem. 3.1.5 for constructor types, Lem. 3.1.13 for application types, (3.1) for arrow types, Lem. 3.1.14 for intersection types and universal quantification, and Cor. 3.1.16 for union types and existential quantification.* □

Type interpretation by reducibility candidates:

$$
\begin{array}{rcl@{\qquad}rcl}
[\alpha]_\rho &=& \rho(\alpha) & [T \cap U]_\rho &=& [T]_\rho \cap [U]_\rho \\
[X]_\rho &=& \rho(X) & [\forall \alpha.U]_\rho &=& \bigcap_{\mathcal{A}\in\mathcal{DC}} [U]_{\rho,\alpha\mapsto\mathcal{A}} \\
[\mathbf{c}]_\rho &=& \overline{\{\mathbf{c}\}} & [\forall X.U]_\rho &=& \bigcap_{\mathcal{A}\in\mathcal{CR}} [U]_{\rho,X\mapsto\mathcal{A}} \\
[DT]_\rho &=& \overline{[D]_\rho\,[T]_\rho} & [T \cup U]_\rho &=& [T]_\rho \cup [U]_\rho \\
[T \to U]_\rho &=& [T]_\rho \to [U]_\rho & [\exists \alpha.U]_\rho &=& \bigcup_{\mathcal{A}\in\mathcal{DC}} [U]_{\rho,\alpha\mapsto\mathcal{A}} \\
 & & & [\exists X.U]_\rho &=& \bigcup_{\mathcal{A}\in\mathcal{CR}} [U]_{\rho,X\mapsto\mathcal{A}}
\end{array}
$$

Type interpretation for case bindings:

$$[\![T]\!]_\rho = \{\, \theta \,/\, \lambda x.\, \{\!|\theta|\!\} \cdot x \in [T]_\rho \}$$

Figure 3.1: Interpretation of types

Notice that we need to use the closure operator to interpret data types. Indeed, for $\mathcal{D} \in \mathcal{DC}$ and $\mathcal{T} \in \mathcal{CR}$, the set $\mathcal{DT}$ does not satisfy (CR$_3$): if $t \in \mathcal{D}$ and $u \in \mathcal{T}$, with both terms in normal form, then the only reduct (assuming $t \neq \maltese$) of the term $\{\!|\mathsf{c} \mapsto tu|\!\} \cdot \mathsf{c}$ is $tu \in \mathcal{DT}$, but $\{\!|c \mapsto tu|\!\} \cdot \mathsf{c}$ itself is not an application, and thus is not in $\mathcal{DT}$. However, this interpretation of types gives a very precise notion of data-types, considering their values.

Proposition 3.2.2. *A term t is a value of $[\mathbf{c}T_1 \dots T_k]_\rho$ iff $t = \mathsf{c}t_1 \dots t_k$ with $t_i \in [T_i]_\rho$ for each i.*

In particular, every $t \in [\mathbf{c}T_1 \dots T_k]_\rho$ is a perfectly normalising closed term, and Prop. 3.1.8 insures that it reduces on a value or on the Daimon. An immediate consequence is that $t \in [\mathbf{c}T_1 \dots T_k]_\rho$ implies

$$t \to^* \mathsf{c}t_1 \dots t_k \text{ for some } t_i \in [T_i]_\rho \ {\scriptstyle (i\leq k)} \qquad \text{or} \qquad t \to^* \maltese$$

Proof : *We show the implication, as the converse is straightforward from an induction on k. We proceed by induction on k. If $k = 0$, it is straightforward from the definition of $[\mathbf{c}]_\rho$.*
Else $[\mathbf{c}T_1 \dots T_k]_\rho = \overline{[\mathbf{c}T_1 \dots T_{k-1}]_\rho [T_k]_\rho}$, and

$$\mathcal{V}(\,[\mathbf{c}T_1 \dots T_k]_\rho\,) \quad = \quad \mathcal{V}(\,[\mathbf{c}T_1 \dots T_{k-1}]_\rho\,[T_k]_\rho\,)$$

So, if t is a value of $[\mathbf{c}T_1 \dots T_k]_\rho$ it is on the form uu' with $u \in [\mathbf{c}T_1 \dots T_{k-1}]_\rho$ and $u' \in [T_k]_\rho$. Moreover, if uu' is a value, it is necessarily a data structure, and u also is a data structure. Hence u is a value of $[\mathbf{c}T_1 \dots T_{k-1}]_\rho$. By induction hypothesis $u = \mathsf{c}t_1 \dots t_{k-1}$ with $t_i \in [T_i]_\rho$, and we conclude with $t_k = u' \in [T_k]_\rho$. □

Corollary 3.2.3. *For any constructor c and any types $T_1, \dots, T_k$,*

$$[\mathbf{c}T_1 \dots T_k]_\rho \;=\; \overline{\mathbf{c}[T_1]_\rho \dots [T_k]_\rho}\ .$$

PROOF : *By Prop. 3.2.2,* $\mathcal{V}([\mathbf{c}T_1 \dots T_k]_\rho) = \mathbf{c}[T_1]_\rho \dots [T_k]_\rho$. *Since the closure operator only adds neutral terms to a set,* $\mathcal{V}(\overline{\mathbf{c}[T_1]_\rho \dots [T_k]_\rho})$ *also is* $\mathbf{c}[T_1]_\rho \dots [T_k]_\rho$, *and Cor. 3.1.10 entails the equality.* □

Soundness w.r.t. sub-typing

Within this interpretation of types, the sub-typing relation becomes the usual sub-set relation.

Lemma 3.2.4. *For any types* T_1, T_2, *if* $T_1 \preccurlyeq T_2$ *then for any valuation* ρ, $[T_1]_\rho \subseteq [T_2]_\rho$.

PROOF : *By induction on the derivation of* $T_1 \preccurlyeq T_2$. *Rules* **Refl** *and* **Trans** *are trivial. Union and intersection rules are straightforward from the definition. Introduction and elimination rules for quantifiers* $\forall$ *and* $\exists$ *use the equality* $[T]_{\rho,\nu\mapsto[U]_\rho} = [T\{U/\nu\}]_\rho$.
Arrow *is standard, and* **Discr** *comes from Proposition 3.2.2:* $[\mathbf{c}_1\vec{T}]_\rho \cap [\mathbf{c}_2\vec{U}]_\rho$ *has no value if* $\mathsf{c}_1 \neq \mathsf{c}_2$ *and thus is smallest than any candidate.*
We detail rules **App** *and* **Data**, *the other rules (commutation rules for type operator) are easy to check: we actually introduced them in the calculus because they were valid in the model (see page 26).*

App: $\dfrac{D \preccurlyeq D' \quad T \preccurlyeq T'}{DT \preccurlyeq D'T'}$

Remark that $\mathcal{D} \subseteq \mathcal{D}'$ *and* $\mathcal{T} \subseteq \mathcal{T}'$ *imply* $\mathcal{D}\mathcal{T} \subseteq \mathcal{D}'\mathcal{T}'$, *and notice that the closure operator is monotonic on sets of terms.*

Data: $D \preccurlyeq T \to DT$

Let ρ *a valuation and* $t \in [D]_\rho$. *Now choose* $u \in [T]_\rho$. *Then* $tu \in [D]_\rho[T]_\rho$, *and this set is included in* $\overline{[D]_\rho[T]_\rho} = [DT]_\rho$. *Hence* $tu \in [DT]_\rho$ *for all* u *in* $[T]_\rho$, *so* $t \in [T \to DT]_\rho$. □

Inhabitants of bottom

In *System F*, type *False* is represented by $\forall X.X$. In our type system, there is the same *False* type, but also a *False* data-type $\forall\alpha.\alpha$. Since every data-type is a type, $\forall X.X \preccurlyeq \forall\alpha.\alpha$ is, unsurprisingly, derivable:

$$\dfrac{\dfrac{}{\forall X.X \preccurlyeq \alpha}\ \forall\textbf{-elim}}{\forall X.X \preccurlyeq \forall\alpha.\alpha}\ \forall\textbf{-intro}$$

In the reducibility model these both types are actually identified:

$$[\forall\alpha.\alpha]_\rho = [\forall X.X]_\rho = \overline{\emptyset}\ .$$

Indeed, $\overline{\emptyset}$ is included in every reducibility candidates (by (CR_3)), so $\overline{\emptyset} \subseteq [\forall\alpha.\alpha]_\rho$. Moreover, $[\forall\alpha.\alpha]_\rho \subseteq [\mathbf{c}]_\rho \cap [\mathbf{c}']_\rho$ (where c and c' are any two different constructors), and $[\mathbf{c}]_\rho \cap [\mathbf{c}']_\rho$ is a reducibility candidates with no value. Hence $[\forall\alpha.\alpha]_\rho \subseteq \overline{\emptyset}$.

Every term of $\overline{\emptyset}$ is perfectly normalising, so it reduces on a normal form in $\overline{\emptyset}$. By Prop. 3.1.8, it necessarily reduces on the Daimon. In fact the semantics of *False* type (and *False* data-type) contains all perfectly normalising terms that are what is sometimes called *hereditarily neutral*: terms that never reduces on a value.

3.2.2 Soundness

In this section we prove the adequacy of the model: if a $\lambda_{\mathscr{C}}$-term has type T, then it belongs to the interpretation of T (and thus is perfectly normalising).

The reducibility candidates model deals with closed terms, whereas proving the adequacy lemma by induction requires the use of open terms —with some assumptions on their free variables, that will be guaranteed by a context. Therefore we use *substitutions* σ, τ to close terms and case bindings M:

$$\sigma, \tau \ := \quad \emptyset \mid x \mapsto u; \sigma \qquad\qquad M_\emptyset = M; \quad M_{x\mapsto u;\sigma} = M[x := u]_\sigma,$$

We complete the interpretation of types with the one of judgements: given a context Γ, we say that a substitution σ *satisfies* Γ for the valuation ρ (notation: $\sigma \in [\Gamma]_\rho$) when

$$(x : T) \in \Gamma \text{ implies } \sigma(x) \in [T]_\rho \ .$$

A typing judgement $\Gamma \vdash t : T$ (or $\Gamma \vdash \theta : T$) is said to be *valid* (notation: $\Gamma \vDash t : T$ or $\Gamma \vDash \theta : T$ respectively) if for every valuation ρ and every substitution $\sigma \in [\Gamma]_\rho$,

$$t_\sigma \in [T]_\rho \qquad\qquad (resp.\, \theta_\sigma \in [\![T]\!]_\rho)$$

The proof of adequacy requires a kind of inversion lemma for $\mathcal{CR}$. Remember that $Red_*(t)$ denotes the set of all reducts (in any number of steps) of a term t.

Lemma 3.2.5. *For any $\mathcal{A} \in \mathcal{CR}$, any terms t, u, and every non-empty non-expansed candidate S,*

$$tu \in \mathcal{A} \quad \Leftrightarrow \quad t \in Red_*(u) \to \mathcal{A} \tag{3.2}$$

$$\lambda x.t \in S \to \mathcal{A} \quad \Leftrightarrow \quad \text{for all } s \in S,\ t[x := s] \in \mathcal{A} \tag{3.3}$$

PROOF : (3.2) *If $tu \in \mathcal{A}$ then for any $u' \in Red_*(u)$, $tu \to^* tu'$, hence $tu' \in \mathcal{A}$ by (CR'_2). So $t \in Red_*(u) \to \mathcal{A}$.*
Conversely, if $t \in Red_*(u) \to \mathcal{A}$ then $tu \in \mathcal{A}$ since $u \in Red_*(u)$.

(3.3) *If $\lambda x.t \in S \to \mathcal{A}$, then for any $s \in S$, $(\lambda x.t)s \in \mathcal{A}$, so $(\lambda x.t)s \to t[x := s]$ implies $t[x := s] \in \mathcal{A}$ by (CR_2). Now, if $t[x := s] \in \mathcal{A}$ for some $s \in S$, then $t \in PN_o$ by Lem. 2.1.2. Moreover, for any $s' \in S$, we can easily check by induction on the reduction of t and s' that $(\lambda x.t)s' \in \mathcal{A}$: it is in $\mathscr{N}_D$, and all its reducts are in $\mathcal{A}$.* □

Remark 3.2.1 . If $u \in PN_o$, then $Red_*(u)$ is a non-expansed candidate, and so $Red_*(u) \to \mathcal{A} \in \mathcal{CR}$ by Lem. 3.1.12. Also, if $u_i \in PN_o$ for $1 \le i \le k$, then

$$t\, u_1 \dots u_k \in \mathcal{A} \quad \Leftrightarrow \quad t \in Red_*(u_1) \to \dots \to Red_*(u_k) \to \mathcal{A}$$

directly results from (3.2) and an induction on k.

The following lemma will help proving the correctness of the typing rule **Cb** in the model:

$$\textbf{Cb}\ \frac{\Gamma \vdash u : \vec{U} \to T \qquad \theta_{\mathsf{d}} \text{ typable for each } \mathsf{d} \in \mathrm{dom}(\theta)}{\Gamma \vdash \theta : \mathbf{c}\vec{U} \to T}\ {\scriptstyle (\mathsf{c} \mapsto u \in \theta)}$$

Lemma 3.2.6. *Let $\mathcal{A}_1, \dots, \mathcal{A}_k, \mathcal{B} \in \mathcal{CR}$ and $\theta \in PN_o$. Assume $\mathsf{c} \mapsto u \in \theta$, with $u \in \vec{\mathcal{A}} \to \mathcal{B}$ (where $\vec{\mathcal{A}} = \mathcal{A}_1; \dots; \mathcal{A}_k$). Then*

$$t \in \overline{\mathbf{c}\mathcal{A}_1 \dots \mathcal{A}_k} \quad \Longrightarrow \quad \{\!|\theta|\!\} \cdot t \in \mathcal{B}$$

PROOF : *We prove that for all $\theta \in PN_o$ with $\mathsf{c} \mapsto u \in \theta$ and $u \in \vec{\mathcal{A}} \to \mathcal{B}$, and for all $t \in \overline{\mathbf{c}\mathcal{A}_1 \dots \mathcal{A}_k}$, the term $\{\!|\theta|\!\} \cdot t$ is in $\mathcal{B}$.*
If t is a value then $t = \mathsf{c}t_1 \dots t_k$ with $t_i \in \mathcal{A}_i$, so

$$\begin{array}{lll} \{\!|\theta|\!\} \cdot t \in \mathcal{B} & \textit{iff} & (\{\!|\theta|\!\} \cdot \mathsf{c})t_1 \dots t_k \in \mathcal{B} \qquad (\mathrm{CR}'_2), (\mathrm{CR}'_4) \\ & \textit{iff} & \{\!|\theta|\!\} \cdot \mathsf{c} \in Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B} \qquad (\textit{Rem. 3.2.1}) \end{array}$$

But $Red_*(t_i) \subseteq \mathcal{A}_i$, so $\mathcal{A}_1 \to \dots \to \mathcal{A}_k \to \mathcal{B} \subseteq Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$. Moreover an immediate induction on the reduction of θ ensures that $\{\!|\theta|\!\} \cdot \mathsf{c}$ is in $\vec{\mathcal{A}} \to \mathcal{B}$: this term is in $\mathscr{N}_D$ and its reducts are either $\{\!|\theta'|\!\} \cdot \mathsf{c}$ with $\theta \to \theta'$ (that is in $\vec{\mathcal{A}} \to \mathcal{B}$ by induction hypothesis), or u (that is in $\vec{\mathcal{A}} \to \mathcal{B}$ by hypothesis). So $\{\!|\theta|\!\} \cdot \mathsf{c}$ is in $\vec{\mathcal{A}} \to \mathcal{B}$ by (CR_3), thus it belongs to $Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$ and so $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$.

Now assume t is neutral. It has the form $ht_1 \dots t_n$ with $h = \maltese$ or $h = \{\!|\phi|\!\} \cdot h_0$ and $n \geq 0$, or $h = \lambda x.h_0$ and $n \geq 1$. We prove that $\{\!|\theta|\!\} \cdot t$ is in $\mathcal{B}$ by induction on the reductions of θ and t.

1. *First consider the cases where $h =?$ or $\{\!|\phi|\!\} \cdot h_0$, and $n \geq 0$:*
$$\begin{array}{lll} \{\!|\theta|\!\} \cdot t \in \mathcal{B} & \textit{iff} & (\{\!|\theta|\!\} \cdot h)t_1 \dots t_n \in \mathcal{B} \qquad (\mathrm{CR}'_2), (\mathrm{CR}'_4) \\ & \textit{iff} & \{\!|\theta|\!\} \cdot h \in Red_*(t_1) \to \dots \to Red_*(t_n) \to \mathcal{B} \qquad (\textit{Rem. 3.2.1}) \end{array}$$
Note that $\{\!|\theta|\!\} \cdot h \in \mathscr{N}_D$ and $Red_*(t_1) \to \dots \to Red_*(t_n) \to \mathcal{B}$ is a reducibility candidate by Lem. 3.1.12. So it is sufficient to show that it contains all reducts of $\{\!|\theta|\!\} \cdot h$. They are either $\maltese$, or $\{\!|\theta'|\!\} \cdot h'$ with $\theta \to \theta'$ and $h = h'$ or $h \to h'$ and $\theta = \theta'$. The Daimon is in every reducibility candidate, and $\{\!|\theta'|\!\} \cdot h' \in Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$ by induction hypothesis. So $\{\!|\theta|\!\} \cdot h \in Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$ by (CR_3), and $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$.

2. *Now consider the case where $h = \lambda x.h_0$ (with $x \notin \mathrm{fv}(\theta)$), and $n \geq 1$.*
$$\begin{array}{lll} \{\!|\theta|\!\} \cdot t \in \mathcal{B} & \textit{iff} & (\lambda x.\{\!|\theta|\!\} \cdot h_0)t_1 \dots t_n \in \mathcal{B} \qquad (\mathrm{CR}'_2), (\mathrm{CR}'_4) \\ & \textit{iff} & \lambda x.\{\!|\theta|\!\} \cdot h_0 \in Red_*(t_1) \to \dots \to Red_*(t_n) \to \mathcal{B} \qquad (3.2) \\ & \textit{iff} & \textit{for all } s \in Red_*(t_1), \\ & & \quad \{\!|\theta|\!\} \cdot h_0[x := s] \in Red_*(t_2) \to \dots \to Red_*(t_k) \to \mathcal{B} \qquad (3.3) \end{array}$$
Furthermore, for any $s \in Red_*(t_1)$, $t \to^* (\lambda x.h_0)\, s\, t_2 \dots t_n \to h_0[x := s]\, t_2 \dots t_n$; thus $\{\!|\theta|\!\} \cdot (h_0[x := s]t_2 \dots t_n) \in \mathcal{B}$ by induction hypothesis.
Hence, $(\{\!|\theta|\!\} \cdot h_0[x := s])t_2 \dots t_n \in \mathcal{B}$ by (CR'_2), and thus by (3.2), $\{\!|\theta|\!\} \cdot h_0[x := s]$ belongs to $Red_*(t_2) \to \dots \to Red_*(t_k) \to \mathcal{B}$. Also $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$.

Finally, $\{\!|\theta|\!\} \cdot t$ always belongs to $\mathcal{B}$. □

Adequacy lemma We finally prove that every derivable judgment is valid.

Proposition 3.2.7 (Adequacy)**.** *Given a term t, a case binding θ, a context Γ and a type T,*

$$\Gamma \vdash t : T \quad \Rightarrow \quad \Gamma \vDash t : T \tag{3.4}$$
$$\Gamma \vdash \theta : T \quad \Rightarrow \quad \Gamma \vDash \theta : T \tag{3.5}$$

PROOF : *The proof is proceeds by induction on the derivation of $\Gamma \vdash t : T$ or $\Gamma \vdash \theta : T$. If the judgement is introduced by the rule* **Init**, **False** *(remember that $\maltese$ is in every reducibility candidate) or* **Constr**,*then it is obvious. If it comes from $\to$* **elim** *it is a direct consequence of the definition of arrow in $\mathcal{CR}$, and the case $\to$* **intro** *is a consequence of (3.3).*
If it comes from **Inter**, **Union**, *or* **Univ** *it is straightforward from induction hypothesis. If it comes from* **Subs**, *it is a consequence of Lem. 3.2.4. We detail the proof in case the derivation comes from rule* **CB** *or* **Exist** *(***Inter** *is similar to this last one).*

Cb: $\dfrac{(\Gamma \vdash u_j : \vec{U_j} \to T_j)_{j=1}^n}{\Gamma \vdash \theta : \mathbf{c}_i\vec{U_i} \to T_i}$ *with* $\theta = \{\mathbf{c}_j \mapsto u_j \,/\, 1 \leq j \leq n\}$

Remember that the interpretation of a type T, seen as a type for case bindings is $[\![T]\!]_\rho = \{\theta \;/\; \lambda x.\{\!|\theta|\!\} \cdot x \in [T]_\rho\}$. Note $(U_{i1} \dots U_{ik}) = \vec{U_i}$, choose ρ a valuation and $\sigma \in [\Gamma]_\rho$, and show that $\lambda x.\{\!|\theta_\sigma|\!\} \cdot x \in [\mathbf{c}_i\vec{U_i} \to T_i]_\rho$. Let $t \in [\mathbf{c}_i\vec{U_i}]_\rho$. By induction on the reduction of θ_σ and t, we show that $(\lambda x.\{\!|\theta_\sigma|\!\} \cdot x)t \in [T_i]_\rho$. This is a neutral term, so it is sufficient to show that all its reducts are in $[T_i]_\rho$. Thanks to induction hypothesis we just have to consider the reduct $\{\!|\theta_\sigma|\!\} \cdot t$.
By Cor. 3.2.3, $t \in \overline{\mathbf{c}_i[U_{i1}]_\rho \dots [U_{ik}]_\rho}$, and $\Gamma \vdash u_i : \vec{U_i} \to T_i$ implies $u_{i\sigma} \in [U_{i1}]_\rho \to \dots \to [U_{ik}]_\rho \to [T_i]_\rho$ by induction hypothesis. All terms in θ_σ are perfectly normalising, so we can use Lem. 3.2.6 to get $\{\!|\theta_\sigma|\!\} \cdot t \in [T_i]_\rho$. Hence $\lambda x.\{\!|\theta_\sigma|\!\} \cdot x \in [\mathbf{c}_i\vec{U_i} \to T_i]_\rho$, wich means $\theta_\sigma \in [\![\mathbf{c}_i\vec{U_i} \to T_i]\!]_\rho$.

Exist: $\dfrac{\Gamma, x : T \vdash t : U}{\Gamma, x : \exists\nu.T \vdash t : U}{}^{\nu \notin \mathrm{TV}(U)}$

Choose a valuation ρ, and a substitution $\sigma \in [\Gamma, x : \exists\nu.T]_\rho$.
Then $\sigma(x) \in \bigcup_{\mathcal{A} \in \mathcal{CR}} [T]_{\rho,\nu\mapsto\mathcal{A}}$. Hence there is some $\mathcal{A} \in \mathcal{CR}$ such that $\sigma(x) \in [T]_{\rho,\nu\mapsto\mathcal{A}}$. Also $\sigma \in [\Gamma, x : T]_{\rho,\nu\mapsto\mathcal{A}}$. By induction hypothesis, $(\Gamma, x : T) \vDash t : U$, so $t_\sigma \in [U]_{\rho,\nu\mapsto\mathcal{A}}$. Since $\nu \notin \mathrm{TV}(U)$, it means that $t_\sigma \in [U]_\rho$. □

Remark 3.2.2 . For a closed term t and a closed type T we immediately get

$$[T] \in \mathcal{CR}\ , \qquad \text{and} \qquad \vdash t : T \ \Rightarrow\ t \in [T]\ .$$

3.2.3 Perfect normalisation without CaseCase

Remembering that reducibility candidates are included in PN_o, an immediate consequence of Rem. 3.2.2 is the perfect normalisation of typed $\lambda^-_{\mathscr{C}}$-calculus.

Theorem 3.1. *Every well typed term is perfectly normalising for $\lambda^-_{\mathscr{C}}$.*

Furthermore, every closed and defined normal form is a value or the Daimon (Prop. 3.1.8). Since the Daimon is never created by a reduction step, typing a term with no subterm ✠ ensures that it strongly reduces —and without case composition— to a value. We can even be more precise concerning data types: if a term (written without ✠) has type $\mathbf{c}T_1 \dots T_k$, then it reduces on a data structure $\mathsf{c}t_1 \dots t_k$ (Prop. 3.2.2).

Now let us call a *pure value* a data-structure whose all sub-terms are data-structures (such as $\mathsf{cons}\ \mathsf{0}\ (\mathsf{cons}\ (\mathsf{S}(\mathsf{S0}))\ \mathsf{nil})$ for instance) and a *pure data type* a data type whose all sub-types are data-types.

A pure value is trivially typable by a pure data-type (just replace every constructor c in the term by the corresponding type constructor $\mathbf{c}$ to obtain the type, and use **Constr** and **Data** to derive the typing judgement). Conversely, every closed defined normal term without ✠ in a pure data type is a pure value (by induction on the structure of the term, using Prop. 3.2.2).

Hence, if t is a term written without the Daimon, and D is a pure data type,

$$\vdash t : D \qquad \Longrightarrow \qquad t \text{ strongly reduces in } \lambda^{-}_{\mathscr{C}} \text{ on a pure value of } D$$

(where a pure value of $\mathbf{c}D_1 \dots D_k$ has form $\mathsf{c}v_1 \dots v_k$ with v_i a pure value of D_i).

In that sense, we can say that case composition is unessential in this calculus: it is not necessary to reach pure values.

Conclusion and future work

The reducibility model we have presented here ensures all the properties we expected for the type system proposed in previous chapter. It is a syntactical model and we conjecture that the typed $\lambda^{-}_{\mathscr{C}}$-calculus is complete for this model, in the sense that if a term t is in the interpretation of a type T, then $\vdash t : T$ is derivable. In Chap. 5 we construct a non-syntactic model for the untyped lambda calculus with constructors in a category of domains.

However we did not deal with the issue of a non-syntactical model for the typed (restricted) $\lambda_{\mathscr{C}}$-calculus. In particular, the semantic meaning of type application remains to be established (even in the realisability model its interpretation is not straightforward, and we have to use a closure operator). More precisely, it would be interesting to define a notion of application types in the categorical setting.

Chapter 4

Categorical model

Denotational semantics aims to interpret programs (including programs that do not have normal form) by some values (the *denotations*). It is not concerned with the computational behaviour of a program, but with its extensional properties. The denotations are objects of some mathematical structure. The most basic one is the universe of sets and functions, but more structured classes such as lattices or topologies are often considered. Category theory outlines an abstract setting to describe those mathematical structures. It provides elementary tools to express what are the essential properties that a model must satisfy.

In Chap. 3, we have defined a denotational model (of reducibility candidates) for the typed lambda calculus with constructors, whom the perfectly normalising theorem resulted from. Here, we establish a notion of a model for the untyped $\lambda_{\mathscr{C}}$-calculus. We define it in a categorical framework, so that it can be instanciated in different "concrete" mathematical structures.

We first recall briefly the categorical notions that we may need in the course of the chapter. Then we formalise what we call a *$\lambda_{\mathscr{C}}$-model*, and we explain how to interpret the $\lambda_{\mathscr{C}}$-terms in such a model. We also prove that this interpretation is *sound*, in the sense that two $\lambda_{\mathscr{C}}$-equivalent terms receive the same denotation. Finally we may present a $\lambda_{\mathscr{C}}$-model (the PER-model) that is *complete* for terms with no match failure, in the sense that it attributes different denotations to defined terms that are not extensionally equal.

4.1 A quick introduction to categories

Category theory is predicated on the idea that many properties of different mathematical constructions can be expressed in a unified way. Concepts of different fields of mathematics (such as Set theory, as well as Topology among others) are abstracted and enunciated with *objects* (that are nothing more than vertexes of a graph) and *arrows*. Category theory is then a study of how these objects and morphisms are structured all together.

It appears that it also proposes an appropriate setting for situations that concern computer scientists. In this thesis we only need very basic notions of

category theory, that we introduce in this section. The interested reader might refer to the Mac Lane's monograph [Lan71] for a more in-depth presentation.

4.1.1 Definitions and examples

A category $\mathbb{C}$ is given by:

- A collection of *objects* $\mathrm{Obj}(\mathbb{C})$ written A, B, C *etc.*
- For each pair of objects (A, B), a collection of *morphisms* $\mathbb{C}[A, B]$ written f, g, h, *etc.*
  We may write $f : A \to B$ when f is a morphism of $\mathbb{C}[A, B]$.
- For each object A, an *identity morphism* $Id_A : A \to A$.
- For each objects A, B, C, a *composition* operation: if $f : A \to B$ and $g : B \to C$ then $f\,;\,g : A \to C$. This composition is associative and has identity morphisms as neutral elements:

$$f\ ;\ (g\ ;\ h) = (f\ ;\ g)\ ;\ h \qquad\qquad f\ ;\ Id_B = f = Id_A\ ;\ f$$

Example 4.1.1 (Category of sets) . $\mathbb{S}\mathrm{et}$ is the category whose objects are sets, and such that $\mathbb{S}\mathrm{et}[A, B]$ is the set of functions from A to B. Identity morphisms and composition are the usual identity functions and composition of functions.

Objects and morphisms of a category $\mathbb{C}$ are usually represented with a graph (a *graph* is just a family of vertexes with a family of arrows between them): a vertex represents an object, and an arrow from A to B represents a morphism of $\mathbb{C}[A, B]$. We call *diagram* the interpretation of a graph in a category. A *path* from A to B in a graph is a sequence of arrows

$$A = A_1 \xrightarrow{f_1} A_2 \xrightarrow{f_2} A_3 \qquad \cdots \qquad A_{n-1} \xrightarrow{f_{n-1}} A_n = B\ .$$

The interpretation of a path in a category is the composition of the morphisms represented by each edge. Also the path above, seen as a diagram, is the morphism $f_1\ ;\ f_2\ \cdots\ ;\ f_{n-1}$. We say that a diagram *commutes* when each path from A to B represents the same morphism, for any vertexes A and B.

Example 4.1.2 . In the category $\mathbb{S}\mathrm{et}$, let Nat and Bool be the sets of natural numbers and of Booleans respectively. Write S the successor function on natural numbers, *even* the function that says whether a natural number in argument is even or not, and *neg* the negation function on Booleans. Then the following diagram commutes:

$$\begin{array}{ccc} \mathrm{Nat} & \xrightarrow{even} & \mathrm{Bool} \\ {\scriptstyle S}\downarrow & & \downarrow{\scriptstyle neg} \\ \mathrm{Nat} & \xrightarrow[even]{} & \mathrm{Bool} \end{array}$$

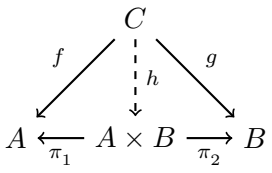
Isomorphisms An *isomorphism* between A and B is given by two morphisms $f : A \to B$ and $g : B \to A$ such that $f \,;\, g = Id_A$ and $g \,;\, f = Id_B$. When such an isomorphism exists we say that A and B are isomorphic, what we write $A \cong B$, or, to make the isomorphism explicit:

$$A \underset{g}{\overset{f}{\cong}} B \ .$$

To be isomorphic is an equivalence relation, also we may sometimes consider objects up to isomorphisms.

4.1.2 Cartesian closed category

Products The categorical product is a generalisation of the Cartesian product of sets. The *product* of two objects A and B of a category $\mathbb{C}$ is given by an object $A \times B$ of $\mathbb{C}$ and two morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ (respectively called the first and second *projection* of $A \times B$) such that, for any object C with morphisms $f : C \to A$ and $g : C \to B$, there exists a unique morphism $h : C \to A \times B$ such that the following diagram commutes:



The morphism h is the only one that makes the diagram commutes, that is why we draw it with a dashed line. We call it the *pairing* of f and g, and we denote it by $\langle f, g \rangle$. Notice that because the uniqueness of h, the product of two objects is unique up to isomorphism.

We can recognise the usual properties of the projections of Cartesian product:

$$f = \langle f, g \rangle \,;\, \pi_1 \qquad g = \langle f, g \rangle \,;\, \pi_2 \qquad h = \langle (h; \pi_1) \,,\, (h; \pi_2) \rangle$$

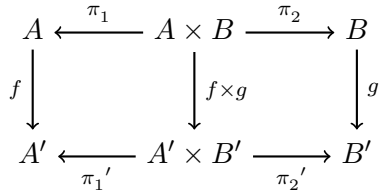
(The last equation results form the uniqueness of pairing, and is called the *surjective pairing* property).

Notice that the categorical product is associative and commutative up to isomorphisms:

$$(A \times B) \times C \underset{\langle \langle \pi_1 ; (\pi_2 ; \pi_1) \rangle \,,\, (\pi_2 ; \pi_2) \rangle}{\overset{\langle (\pi_1 ; \pi_1) \,,\, \langle (\pi_1 ; \pi_2) \,,\, \pi_2 \rangle \rangle}{\cong}} A \times (B \times C) \qquad A \times B \underset{\langle \pi_2 \,,\, \pi_1 \rangle}{\overset{\langle \pi_2 \,,\, \pi_1 \rangle}{\cong}} B \times A$$

If A and B have a product, and A' and B' also, then for any $f : A \to B$ and $g : A' \to B'$ we write $f \times g$ the morphism $\langle (\pi_1 ; f) \,,\, (\pi_2 ; g) \rangle$. It is called the *product* of f and g, and it is the unique morphism such that the following

diagram commutes:



Generalised product The definition of the categorical product can be easily extended to generalised products. A family of objects $(A_i)_{i\in I}$ has a product if there is an object $\Pi_{i\in I}A_i$ and a family of projection morphisms $\pi_j : \left(\Pi_{i\in I}A_i\right) \to A_j$ indexed by I such that for any object C with morphisms $f_i : C \to A_i$, there is a unique morphism $h : C \to \Pi_{i\in I}A_i$ such that $f_i = h;\pi_i$ for all $i \in I$.

- *Finite product:* If $I = [\![1..n]\!]$, then $\Pi_{i\in I}A_i$ is called the n-ary product of $A_1, \ldots, A_{n-1}$ and A_n. We might write it $A_1 \times \ldots \times A_n$, or A^n if all the A_i's are the same object A. We also write $\pi_i^n : A_1 \times \ldots \times A_n \to A_i$ the i^{th} projection morphisms.
- *Terminal object:* If I is the empty set, then $\Pi_{i\in I}A_i$ is written $\mathbf{1}$, and is called a *terminal* object. It is the unique object (up to isomorphism) such that for any C there is a unique morphism $!_C : C \to \mathbf{1}$.

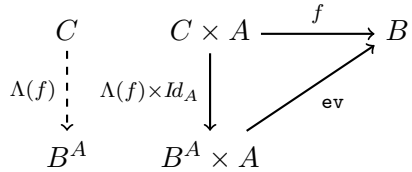
Definition 4.1.1 (*Cartesian category*)

A category $\mathbb{C}$ is Cartesian if it has a terminal object $\mathbf{1}$ and if, for any objects A and B the product of A and B exists.
In such a category, there is a unique morphism $!_A$ in $\mathbb{C}[A, \mathbf{1}]$ for each object A, and a morphism in $\mathbb{C}[\mathbf{1}, A]$ is called a *point* of A.

Remark 4.1.3 . A category is Cartesian *iff* it has all finite products of objects. Indeed, in the same way that we showed the associativity up to isomorphism, we can decompose any n-ary product in binary products:

$$A_1 \times A_2 \times \cdots \times A_n \cong A_1 \times \big(A_2 \times (\cdots \times A_n)\big)$$

Exponent In a Cartesian category $\mathbb{C}$, the *exponent* of to objects A and B is given by an object B^A and a morphism $\mathtt{ev} : B^A \times A \to B$ such that, for any object C with a morphism $f : C \times A \to B$, there is a unique morphism in $\mathbb{C}[C \to B^A]$ (that we write $\Lambda(f)$) such that the following diagram commutes:



The morphism $\Lambda(f)$ is called the *curried form* of f. Since, conversely, the morphism f is uniquely determined from $\Lambda(f)$ (as $f = (\Lambda(f) \times Id)$; $\mathtt{ev}$), we may call it the *uncurried* form of $\Lambda(f)$.

Definition 4.1.2 (*Cartesian Closed Category*)

A Cartesian category $\mathbb{C}$ is *closed* if every pair of objects has an exponent. In that case we say that $\mathbb{C}$ is a CCC.

The category $\mathbb{S}$et is Cartesian closed: for any two sets A and B, the object B^A is the set of functions form A to B, and $\mathtt{ev}$ is the evaluation function.

We give some trivial properties of Cartesian closed categories that will be needed in the following.

Lemma 4.1.1. *If A and A' are isomorphic, as well as B and B', then $A \times A'$ and $B \times B'$ are isomorphic:*

$$A \underset{f'}{\overset{f}{\cong}} B \qquad \textit{and} \qquad A' \underset{g'}{\overset{g}{\cong}} B' \qquad \textit{implies} \qquad A \times A' \underset{f' \times g'}{\overset{f \times g}{\cong}} B \times B'$$

Lemma 4.1.2. *In a* CCC*, for any morphisms $f : A \to A'$, $g : B \to B'$ and $h : C \to C'$ both following diagrams commute:*

$$\begin{array}{ccc} (A \times B) \times C & \xrightarrow{\cong} & A \times (B \times C) \\ {\scriptstyle (f\times g)\times h}\downarrow & & \downarrow{\scriptstyle f\times(g\times h)} \\ (A' \times B') \times C' & \xrightarrow[\cong]{} & A' \times (B' \times C') \end{array} \qquad \begin{array}{ccccc} \mathbf{1} \times A & \xrightarrow{\cong} & A & \xrightarrow{\cong} & A \times \mathbf{1} \\ {\scriptstyle \mathbf{1}\times f}\downarrow & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle f\times\mathbf{1}} \\ \mathbf{1} \times A' & \xrightarrow[\cong]{} & A' & \xrightarrow[\cong]{} & A' \times \mathbf{1} \end{array}$$

Lemma 4.1.3. *In any* CCC*, given four objects A, B, C and C', and three morphisms $g : C \times A \to B$, $g' : C' \times A \to B$ and $h : C \to C'$,*

$$\Lambda(g) = h; \Lambda(g') \iff g = (h \times Id_A); g' \ .$$

Proof : *By uniqueness of the exponent.*

The following proposition corresponds to the *currification* of functions:

Proposition 4.1.4. *In a* CCC*, for any three objects A, B and C,*

$$C^{B\times A} \cong (C^B)^A.$$

4.2 Categorical model of $\lambda_{\mathscr{C}}$

4.2.1 Lambda calculus and CCC

In functional languages, functions are values. They can be either applied or given as argument to other programs. In this sense, functions on programs live

in the same universe as programs themselves. This intuition is well conveyed in Cartesian closed categories: if two objects A and B represent certain classes of programs, then there is also an object B^A representing the functions on the programs of A that return a program of B.

It is well known [AL91, Chap. 8] that Cartesian closed categories have exactly the good structure to interpret the typed lambda calculus. The idea is to interpret a type T by an object A_T in a CCC and a judgement $x_1 : U_1, \ldots, x_k : U_k \vdash t : T$ by a morphism $[t] : A_{U_1} \times \ldots \times A_{U_k} \to A_T$. In particular, a closed term of type T is represented by a *point* of A_T.

In the *untyped* lambda calculus, we artificially add a unique type T for all terms, and we interpret it with an object D. A term t with free variables in $x_1, \ldots, x_k$ is interpreted like the judgement $x_1 : T, \ldots, x_k : T \vdash t : T$ in the typed setting, *i.e.* by a morphism of $D^k \to D$.

Without type, any term can be *applied* to an other one, which means that any point t of D can be seen as a function on terms, *i.e.* as a point of D^D. Let us use the informal notation $\hat{x}.t$ to denotes the function $u \mapsto t[x := u]$. Then we need a morphism $\begin{array}{rcl} D & \to & D^D \\ t & \mapsto & \hat{x}.(tx) \end{array}$ (where x is fresh in t).
In the same way, a substitutive variable in an open term can be bound by a λ-abstraction. Also we need a converse morphism $\begin{array}{rcl} D^D & \to & D \\ \hat{x}.t & \mapsto & \lambda x.t \end{array}$.
Now, if the model *validates* the β-reduction, it means that for any points t, u of D, $(\lambda x.t)u \simeq t[x := u]$. One one hand,

$$(\lambda x.t)u \quad = \quad (\mathtt{app}\ (\lambda x.t))\ (u) \quad = \quad (\mathtt{app}\ (\mathtt{lam}\ (\hat{x}.t)))\ (u)\ .$$

On the other hand, $t[x := u] = \hat{x}.t\ (u)$. Thus the β-reduction requires the equality $\mathtt{lam}\ ;\ \mathtt{app} = Id_{D^D}$. This leads to the notion of *reflexive object*.

Definition 4.2.1 (*Reflexive object*)

> In a Cartesian closed category, an object D is *reflexive* if there are two morphisms $\mathtt{app} : D \to D^D$ and $\mathtt{lam} : D^D \to D$ such that $\mathtt{lam}; \mathtt{app} = Id_{D^D}$.

If in addition $\mathtt{app}; \mathtt{lam} = Id_D$ then D is isomorphic to D^D. In this case the η-rule is valid too. Indeed, for any point t of D with $x \notin \mathrm{fv}(t)$, then

$$\lambda x.(tx) \ = \ \mathtt{lam}(\hat{x}.(tx)) \ = \ \mathtt{lam}(\mathtt{app}(t))\ .$$

Hence $\mathtt{app}; \mathtt{lam} = Id_D$ implies $\lambda x.(tx) \simeq t$ if $t \notin \mathrm{fv}(t)$.

In the next section we constrain a CCC with a bit more structure in order to build a model for the lambda calculus with constructors.

4.2.2 $\lambda_{\mathscr{C}}$-models

From now on, we consider that the set of constructors is finite:

$$\mathscr{C} \ = \ \{\mathtt{c}_1, \ldots, \mathtt{c}_n\}\ .$$

Furthermore, in this chapter we restrict the calculus: we drop out the Daimon in order not to overload the definition of a $\lambda_{\mathscr{C}}$-model. It is possible since the $\lambda_{\mathscr{C}}$-calculus with no Daimon is stable by reduction (*cf.* the paragraph on Daimon rules page 16), and we will see in next chapter that ✠ can be encoded with a term having the same behaviour in pure $\lambda_{\mathscr{C}}$-calculus.

Also in this chapter, $\lambda_{\mathscr{C}}$ denotes the lambda calculus with *no Daimon and no Daimon rules* (but with rule CaseCase).

Like for the pure lambda calculus, a Cartesian closed category $\mathbb{C}$ that is a model for the untyped $\lambda_{\mathscr{C}}$-calculus is provided with an object D isomorphic to D^D. Terms are interpreted by points of D. In particular, each constructor c is interpreted by a special morphism $c^* : \mathbf{1} \to D$.

A case-binding $\theta = \{\mathsf{c}_i \mapsto u_i / 1 \leq i \leq n\}$ is interpreted by a point of D^n. Such a point is informally written $\vec{u}$ or $(u_i)_{i=1}^n$, and we then denotes by $\{\vec{\mathsf{c}} \mapsto \vec{u}\}$ the case-binding θ that it represents. The way we interpret θ if $\mathrm{dom}(\theta) \neq [\![1..n]\!]$ is detailed in the discussion p. 58. In the syntax, the case construct builds a term $\{\!|\theta|\!\} \cdot t$ from a case-binding θ and a term t. It corresponds in the model to this morphism:

$$\begin{array}{rccl} \mathsf{case}: & D^n \times D & \to & D \\ & (\vec{u}, t) & \mapsto & \{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t \end{array}$$

Then $\{\!|\mathsf{c}_i \mapsto u_i / 1 \leq i \leq n|\!\} \cdot \mathsf{c}_i \simeq u_i$ (rule CaseCons) means that $\mathsf{case}\ (\vec{u}, \mathsf{c}_i) \simeq \pi_i^n(\vec{u})$. This amounts to the commutation of the diagram $(D2)$ (Fig. 4.1). In the same way, the rule CaseCase is valid if the diagram $(D5)$ commutes, *i.e.* if

$$\begin{array}{ccccccc} D^n \times D \times D & \xrightarrow{\_\times \mathsf{app} \times \_} & D^n \times D^D \times D & \xrightarrow{\_\times \mathsf{ev}} & D^n \times D & \xrightarrow{\quad\mathsf{case}\quad} & D \\ (\vec{u}, t, t') & & (\vec{u}\,,\, \hat{x}.tx\,,\, t') & & (\vec{u}\ ,\ tt') & & \{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot (tt') \end{array}$$

is equal to

$$\begin{array}{ccccccc} D^n \times D \times D & \xrightarrow{\mathsf{case} \times \_} & D \times D & \xrightarrow{\quad\mathsf{app} \times \_\quad} & D^D \times D & \xrightarrow{\quad\mathsf{ev}\quad} & D \\ (\vec{u}, t, t') & & (\{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t\ ,\ t') & & \big(\hat{x}.(\{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t)x\ ,\ t'\big) & & (\{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t)\ t' \end{array}$$

To express the rule CaseLam we need a morphism that abstracts the case construct from a variable:

$$\begin{array}{rccl} \mathsf{case}^\circ = \Lambda(f_{\mathsf{case}}): & D^n \times D^D & \to & D^D \\ & (\vec{u}, \hat{x}.t) & \mapsto & \hat{x}.\ \{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t \end{array}$$

where $f_{\mathsf{case}} = \quad (D^n \times D^D) \times D \xrightarrow{\cong} D^n \times (D^D \times D) \xrightarrow{Id_{D^n} \times \mathsf{ev}} D^n \times D \xrightarrow{\mathsf{case}} D$.

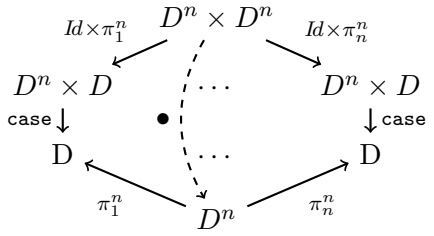
Then the rule CaseLam is valid if $(D4)$ commutes:

$$\begin{array}{ccccccccc} D^n \times D^D & \xrightarrow{\mathtt{case}^\circ} & D^D & \xrightarrow{\mathtt{lam}} & D & = & D^n \times D^{D} \xrightarrow{-\times \mathtt{lam}} D^n \times D & \xrightarrow{\mathtt{case}} & D \\ (\vec{u}, \hat{x}.t) & & \hat{x}.\{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t & & \lambda x.\{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t & & (\vec{u}, \hat{x}.t) & & (\vec{u}, \lambda x.t) \quad \{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot \lambda x.t \end{array}$$

In the same way, the rule CaseCase requires a morphism for the composition of case-bindings:

$$\begin{array}{rccc} \bullet : & D^n \times D^n & \to & D^n \\ & (\vec{u}, (t_i)_{i=1}^n) & \mapsto & (\{\!|\vec{\mathsf{c}} \mapsto \vec{u}|\!\} \cdot t_i)_{i=1}^n \end{array}$$

It is defined as the pairing of the morphisms $(Id_{D^n} \times \pi_i^n); \mathtt{case}$, for $1 \leq i \leq n$. So it is the unique morphism that makes the diagram on the following commute.



Then the commutation of the diagram $(D5)$ validates the rule CaseCase.

Interpretation of incomplete case-bindings. Interpreting a case-binding $\{\mathsf{c}_i \mapsto u_i / 1 \leq i \leq n\}$ like the n-tuple $(u_i)_{i=1}^n$ raises the problem of how to interpret an "incomplete" case-binding θ (with $\mathrm{dom}(\theta) \subsetneq [\![1..n]\!]$). The solution we have chosen here consists in adding a constant $\grave{}$ in the model. Then we interpret θ by the n-tuple $(u_i)_{i=1}^n$, where $u_i = \theta_{\mathsf{c}_i}$ if $\mathsf{c}_i \in \mathrm{dom}(\theta)$, and $u_i = \grave{}$ otherwise.

This is not completely satisfactory, since it leads to unifying some non convertible terms, such as for instance $\{\!|\mathsf{c}_1 \mapsto \lambda x.x|\!\} \cdot \mathsf{c}_2$ and $\{\!|\mathsf{c}_2 \mapsto \lambda x.xx|\!\} \cdot \mathsf{c}_1$: the first term is represented by $\{\!|\lambda x.x, \grave{}|\!\} \cdot \mathsf{c}_2$ (assuming $n = 2$) which is equal to $\grave{}$, and the second one by $\{\!|\grave{}, \lambda x.xx|\!\} \cdot \mathsf{c}_1$ which is also equal to $\grave{}$[1].

Moreover, this design requires an additional equivalence in the model, that corresponds to the propagation of undefined branches of a case-binding while case-commutation. Indeed, if $\mathsf{c}_i \notin \mathrm{dom}(\phi)$, then $\mathsf{c}_i \notin \mathrm{dom}(\theta \circ \phi)$ for any case-binding θ. Also $\theta \circ \phi$ should be represented by a n-tuple whose i^{th} component is $\grave{}$. Yet, $\theta \circ \phi$ is interpreted by a n-tuple whose i^{th} component is $\{\!|\theta|\!\} \cdot u_i$ if $\mathsf{c}_i \mapsto u_i \in \phi$, and $\{\!|\theta|\!\} \cdot \grave{}$ if $\mathsf{c}_i \notin \mathrm{dom}(\phi)$. That is why we need the equivalence $\{\!|\theta|\!\} \cdot \grave{} \simeq \grave{}$ (which corresponds to the commutation of $(D6)$).

[1] We could also have chosen to add n different constants $\grave{}_1, \ldots, \grave{}_n$ in the model, each one to complete only one branch of the case-binding. In that case $\{\!|\mathsf{c}_1 \mapsto \lambda x.x|\!\} \cdot \mathsf{c}_2$ would have receive the denotation $\grave{}_2$, and $\{\!|\mathsf{c}_2 \mapsto \lambda x.xx|\!\} \cdot \mathsf{c}_1$ the denotation $\grave{}_1$. However, $\{\!|\mathsf{c}_1 \mapsto \lambda x.x|\!\} \cdot \mathsf{c}_2$ and $\{\!|\mathsf{c}_1 \mapsto \lambda x.xx|\!\} \cdot \mathsf{c}_2$ would both be interpreted by $\grave{}_2$ whereas they are not $\lambda_{\mathscr{C}}$-equivalent.

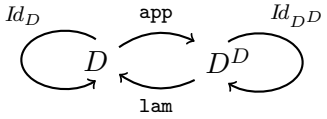
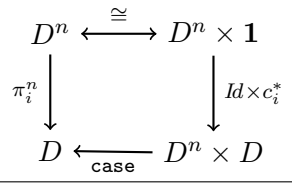
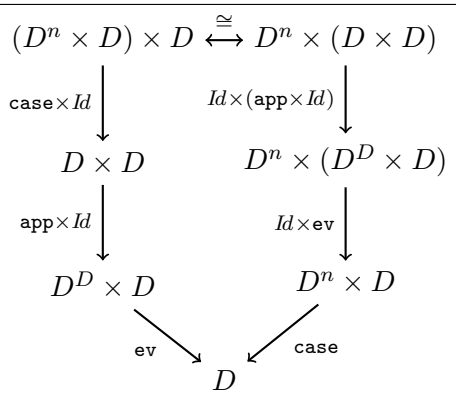
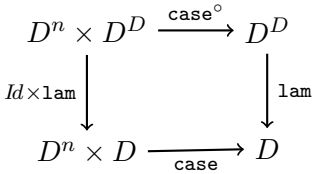
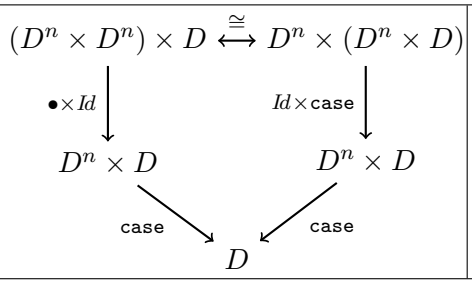
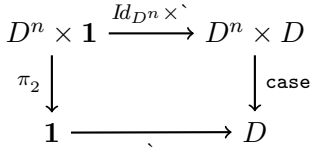
LamApp/AppLam		CaseCons	
$(D1)$		$(D2)$	
CaseApp		**CaseLam**	
$(D3)$		$(D4)$	
CaseCase			
$(D5)$		$(D6)$	

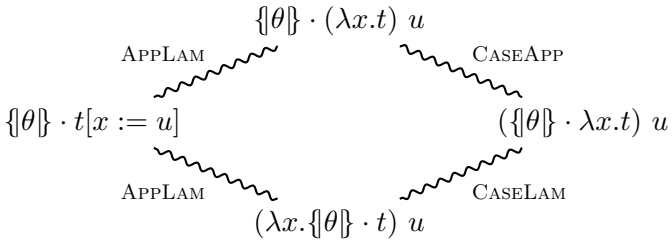
Figure 4.1: Commuting diagrams in a $\lambda_{\mathscr{C}}$-model

Definition 4.2.2 ($\lambda_{\mathscr{C}}$-*model*)

A categorical model for the untyped $\lambda_{\mathscr{C}}$-calculus is $\mathscr{M} = (\mathbb{C}, D, \mathtt{app}, \mathtt{lam}, (c_i^*)_{i=1}^n, \grave{}, \mathtt{case})$ where

- $\mathbb{C}$ is a Cartesian closed category,
- The object D of $\mathbb{C}$ is isomorphic to D^D,
- All the c_i^*'s and $\grave{}$ are points of D,
- $\mathtt{app}$ is a morphism of $D \to D^D$, $\mathtt{lam}$ is a morphism of $D^D \to D$ and $\mathtt{case}$ a morphism of $D^n \times D \to D$,
- The six diagrams of Fig. 4.1 commute ($(D2)$ must commute for every $i \in [\![1..n]\!]$).

Equivalent definition. In fact we can simplify the definition of a $\lambda_{\mathscr{C}}$-model, since the diagrams $(D3)$ and $(D4)$ are equivalent within the isomorphism $D \cong D^D$. This can be understood from a syntactical point of view, given that the commutation of the diagram $(D3)$ validates the rule CaseApp, and the one of $(D4)$ validates CaseLam. Remember that the rule CaseLam was introduced in the calculus in order to close a critical pair (*cf.* Sec. 2.1.3). Thus considering $\lambda_{\mathscr{C}}$-equivalence instead of reduction steps, it entails that CaseApp and CaseLam are redundant in presence of AppLam:

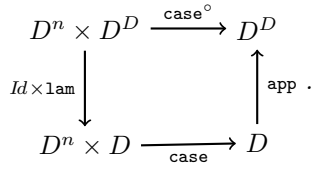


Hence if CaseApp and AppLam are valid then $(\{\!|\theta|\!\} \cdot \lambda x.t)\ u \simeq (\lambda x.\{\!|\theta|\!\} \cdot t)\ u$, and if CaseLam and AppLam are valid then $\{\!|\theta|\!\} \cdot (\lambda x.t)u \simeq (\{\!|\theta|\!\} \cdot \lambda x.t)\ u$.

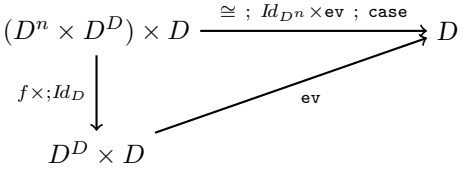
This is formalised in the semantic setting by the following proposition.

Proposition 4.2.1. *If* $\mathtt{lam}$ *and* $\mathtt{app}$ *form an isomorphism between* D *and* D^D*, then the diagram* $(D3)$ *commutes if and only if the diagram* $(D4)$ *commutes.*

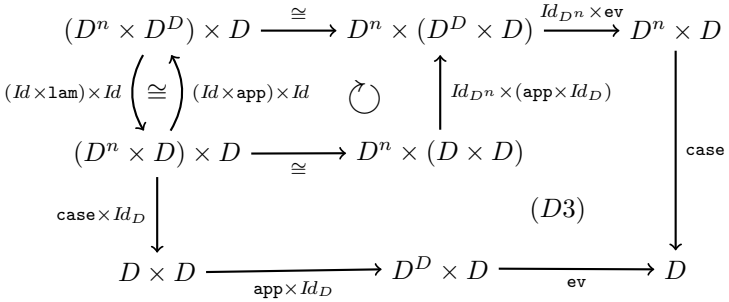
Proof : *Since* $(D1)$ *commutes,* $(D4)$ *commutes iff the following diagram commutes:*



Write $f = Id_{D^n} \times \mathtt{lam}; \mathtt{case}; \mathtt{app}$. *Since* $\mathtt{case}^\circ = \Lambda(\cong; Id_{D^n} \times \mathtt{ev}; \mathtt{case})$, *and by uniqueness of the exponent,* $f = \mathtt{case}^\circ$ *if and only if the following diagram commutes:*



We can detail this diagram as follows:



Since the sub-diagram in the upper-left corner commutes (by Lem. 4.1.2), $(D4)$ *commutes if and only if* $(D3)$ *commutes.* □

So we can omit the commutation of $(D3)$ or the one of $(D4)$ in the definition of a $\lambda_{\mathscr{C}}$-model.

4.2.3 Soundness

Here we formalise the fact that the notion of $\lambda_{\mathscr{C}}$-models we gave in the previous section effectively defines *sound* models for the untyped $\lambda_{\mathscr{C}}$-calculus. Remember that $\simeq_{\lambda_{\mathscr{C}}}$ denotes the reflexive, transitive and symmetric closure of $\to$, the reduction relation in the $\lambda_{\mathscr{C}}$-calculus (with no Daimon).

Theorem 4.1 (Soundness)**.** *If* $\mathscr{M} = (\mathbb{C}, D, \mathtt{lam}, \mathtt{app}, (\mathtt{c}_i^*)_{i=1}^n, \mathtt{case}, \grave{})$ *is a* $\lambda_{\mathscr{C}}$*-model, then we can interpret each closed* $\lambda_{\mathscr{C}}$*-term* t *by a point* $[t]$ *of* D *such that*

$$t \simeq_{\lambda_{\mathscr{C}}} t' \quad \Longrightarrow \quad [t] = [t']$$

The interpretation of a term is defined by structural induction, so we need to consider open terms. Given $\Gamma = x_1, \cdots, x_k$ a list of variables and t a term (or θ a case-binding) whose all free variables are in Γ, $[t]_\Gamma$ is a morphism of $D^k \to D$ (or $[\theta]_\Gamma$ is a morphism of $D^k \to D^n$) defined by induction in Fig. 4.2. If a term t is closed, we then write $[t] = [t]_\emptyset$ its denotation in $\mathbf{1} \to D$. We prove the Theo. 4.1 for all the $\lambda_{\mathscr{C}}$-terms, included the open ones.

Within the framework of this interpretation, morphism $\bullet$ actually corresponds to case-composition, as it is formalised by the following lemma.

$$
\begin{array}{rcl}
[x_i]_\Gamma & = & \pi_i^k : D^k \to D \\
[tu]_\Gamma & = & D^k \xrightarrow{\langle [t]_\Gamma ; [u]_\Gamma \rangle} D \times D \xrightarrow{\mathtt{app} \times Id_D} D^D \times D \xrightarrow{\mathtt{ev}} D \\
[\lambda x_{k+1}.t]_\Gamma & = & D^k \xrightarrow{\Lambda(f_t)} D^D \xrightarrow{\mathtt{lam}} D \\
\text{where } f_t & = & D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t]_{\Gamma, x_{k+1}}} D \\
[\mathsf{c}]_\Gamma & = & D^k \xrightarrow{!_{D^k}} \mathbf{1} \xrightarrow{c^*} D \\
[\{\!|\theta|\!\} \cdot t]_\Gamma & = & D^k \xrightarrow{\langle [\theta]_\Gamma ; [t]_\Gamma \rangle} D^n \times D \xrightarrow{\mathtt{case}} D \\
[\theta]_\Gamma & = \langle f_1 ; \cdots ; f_n \rangle : D^k \to D^n \ , & \text{where } f_i = \left\{ \begin{array}{ll} [u_i]_\Gamma & \text{if } \mathsf{c_i} \mapsto u_i \in \theta \\ !_{D^k} ; \grave{} & \text{if } \mathsf{c_i} \notin \mathrm{dom}(\theta) \end{array} \right.
\end{array}
$$

Figure 4.2: Interpretation of $\lambda_{\mathscr{C}}$-terms in a categorical model

Lemma 4.2.2 (Categorical case-composition)**.** *If the diagram* $(D6)$ *commutes, then for any case-bindings* θ *and* ϕ*, whose free variables are in* $\Gamma = \{x_1, \ldots, x_k\}$*, the following diagram commute:*

$$
\begin{array}{ccc}
D^k & \xrightarrow{\langle [\theta]_\Gamma, [\phi]_\Gamma \rangle} & D^n \times D^n \\
& \searrow^{[\theta \circ \phi]_\Gamma} & \downarrow \bullet \\
& & D^n
\end{array}
\tag{4.1}
$$

Proof : *If* $\phi = \{\mathsf{c_i} \mapsto u_i / i \in J\}$ *(with* $J \subseteq [\![1..n]\!]$*), then*

$$
[\theta \circ \phi]_\Gamma = \langle f_1, \ldots, f_n \rangle \ , \quad \text{with } f_i = \left\{ \begin{array}{ll} [\{\!|\theta|\!\} \cdot u_i]_\Gamma & \text{if } i \in J \\ !_{D^k} ; \grave{} & \text{if } i \notin J \end{array} \right.
$$

On the other hand, $\bullet = \langle \big((Id_{D^n} \times \pi_1^n) ; \mathtt{case}\big), \ldots, \big((Id_{D^n} \times \pi_1^n) ; \mathtt{case}\big) \rangle$*. So*

$$
\langle [\theta]_\Gamma, [\phi]_\Gamma \rangle \ ; \ \bullet = \langle g_1, \ldots, g_n \rangle, \qquad \text{with} \quad g_i = \langle [\theta]_\Gamma, ([\phi]_\Gamma \ ; \ \pi_i^n) \rangle \ ; \ \mathtt{case} \ .
$$

If $i \in J$, $[\phi]_\Gamma \ ; \ \pi_i^n = [u_i]_\Gamma$ *and then* $g_i = \langle [\theta]_\Gamma, [u_i]_\Gamma \rangle \ ; \ \mathtt{case}$ *which is* f_i*.*
If $i \notin J$*, then* $[\phi]_\Gamma \ ; \ \pi_i^n = !_{D^k} \times \grave{}$*. Hence*

$$
\begin{array}{rcl}
g_i & = & D \xrightarrow{\langle [\theta]_\Gamma, !D^k \rangle} D^n \times \mathbf{1} \xrightarrow{Id_{D^n} \times \grave{}} D^n \times D \xrightarrow{\mathtt{case}} D \\
& = & D \xrightarrow{\langle [\theta]_\Gamma, !D^k \rangle} D^n \times \mathbf{1} \xrightarrow{\pi_2} \mathbf{1} \xrightarrow{\grave{}} D \qquad \textit{(by } (D6)\textit{)} \\
& = & D \xrightarrow{!D^k} \mathbf{1} \xrightarrow{\grave{}} D
\end{array}
$$

So $g_i = f_i$ *for any* $i \leq n$, *and* $\langle [\theta]_\Gamma, [\phi]_\Gamma \rangle \ ; \ \bullet \ = \ [\theta \circ \phi]_\Gamma$. □

Sound interpretation The proof of soundness requires some usual preliminary lemmas (corresponding to context manipulation in the typed setting, and to variable substitution).

Lemma 4.2.3 (Contextual rules)**.**
Exchange: Let $\Gamma = \{x_1, \dots, x_k\}$ *and* σ *a substitution over* $[\![1..k]\!]$*. Write* $\sigma(\Gamma) = \{\sigma(1), \dots, \sigma(k)\}$*. Then, for any term* t *whose free variables are in* Γ*,*

$$[t]_\Gamma \;=\; \langle \pi^k_{\sigma(1)}, \dots, \pi^k_{\sigma(k)} \rangle \;;\; [t]_{\sigma(\Gamma)} \;.$$

Weakening: Let $\Gamma = \{x_1, \dots, x_k\}$ *containing all free variables of a term* t*, and* $y \notin \Gamma$*. Then*

$$[t]_{\Gamma,y} = \; \langle \pi^{k+1}_1, \dots, \pi^{k+1}_k \rangle \;;\; [t]_\Gamma \;.$$

Proof : *By structural induction on* t*.* □

Lemma 4.2.4 (Substitution)**.** *Given* $\Gamma = \{x_1, \dots, x_k\}$*, and two terms* t *and* u *such that* $\mathrm{fv}(u) \subseteq \Gamma$ *and* $\mathrm{fv}(t) \subseteq \Gamma \cup \{y\}$*,*

$$[t[y := u]]_\Gamma \;=\; D^k \xrightarrow{\langle Id, [u]_\Gamma \rangle} D^k \times D \xrightarrow{\;\cong\;} D^{k+1} \xrightarrow{[t]_{\Gamma,y}} D$$

Proof : *cf. appendix A.1.1.* □

The soundness theorem is then a direct corollary of the following proposition:

Proposition 4.2.5. *If* $\mathscr{M} = (\mathbb{C}, D, \mathtt{lam}, \mathtt{app}, (\mathsf{c}_\mathsf{i}{}^*)_{i=1}^n, \mathtt{case}, \grave{})$ *is a* $\lambda_{\mathscr{C}}$*model, then for any* $\Gamma = \{x_1, \dots, x_k\}$ *and any terms* t_1, t_2 *such that* $\mathrm{fv}(t_1) \subseteq \Gamma$ *and* $t_1 \to t_2$*, the interpretation given in Fig. 4.2 satisfies* $[t_1]_\Gamma = [t_2]_\Gamma$*.*

Proof : *cf. appendix A.1.1.* □

Remark 4.2.1 . As it was explained in the discussion page 58, in a $\lambda_{\mathscr{C}}$-model, all match failing terms $\{\!|\theta|\!\} \cdot \mathsf{c}$ (with $\mathsf{c} \notin \mathrm{dom}(\theta)$) receives the same interpretation:

$$[\{\!|\theta|\!\} \cdot \mathsf{c}] = \grave{} \;.$$

4.3 Completeness

In this part we shall prove the converse of Theo. 4.1:

Theorem 4.2 (Completeness)**.** *If* t *and* t' *are two hereditarily defined* $\lambda_{\mathscr{C}}$*-terms such that in any categorical* $\lambda_{\mathscr{C}}$*-model* $[t]{=}[t']$*, then*

$$t \simeq_{\lambda_{\mathscr{C}}} t' \;.$$

It means that, without match failure, the diagrams of Fig. 4.1 are minimal. Notice that, because of Rem. 4.2.1, the completeness theorem does not hold for

undefined terms. The completeness result is established using the same method as [FM09]:

1. We define $\mathbb{P}\text{ER}_{\lambda_{\mathscr{C}}}$, the Cartesian closed category of partial equivalence relation compatible with $\simeq_{\lambda_{\mathscr{C}}}$.
2. In this syntactic category, we construct a $\lambda_{\mathscr{C}}$-model $\mathscr{M}_{synt}$.
3. Then we show that if $[t] = [t']$ in $\mathscr{M}_{synt}$, then $t \simeq_{\lambda_{\mathscr{C}}} t'$.

4.3.1 Partial equivalence relations

Partial equivalence relations (PER) are commonly used to transform a model of the untyped lambda calculus into a model of the typed lambda-calculus [Mit86, TC87]. Yet we use them here to instantiate the definition of $\lambda_{\mathscr{C}}$-models in the category of PER on $\lambda_{\mathscr{C}}$-terms. Thereby we construct a syntactic model of the untyped $\lambda_{\mathscr{C}}$-calculus.

Definition 4.3.1 ($\boldsymbol{\lambda_{\mathscr{C}}-per}$)

Given a set X, a *partial equivalence relation* on X is a binary relation R that is symmetric and transitive. We may write $x = y : R$ instead of $(x, y) \in R$. A $\lambda_{\mathscr{C}}-per$ is a partial equivalence relation R on Λ (the set of all $\lambda_{\mathscr{C}}$-terms) that is *compatible* with $\lambda_{\mathscr{C}}$, which means:

$$\begin{cases} t = t' : R \\ t_0 \simeq_{\lambda_{\mathscr{C}}} t' \end{cases} \quad \text{implies} \quad t = t_0 : R$$

Given a partial equivalence relation R, the *class* of an element e modulo R is

$$\overline{e}^R = \{\ e' \,/\, e = e' : R\ \}.$$

An element e is *accessible* by R when its class modulo R is non empty. We may write $e \in R$ when e is accessible by R. Notice that if there is some e' such that $e = e' : R$, then by symmetry and transitivity $e = e : R$. In particular, R is an equivalence relation on the set of its accessible elements. The *domain* of R is the set of its accessible elements *modulo* R:

$$\text{dom}(R) = \{\ \overline{e}^R \,/\, e \in R\ \}$$

Also all the following assertions are equivalent:

$$e = e : R \quad ; \quad e \in R \quad ; \quad \overline{e}^R \neq \emptyset \quad ; \quad \overline{e}^R \in \text{dom}(R)$$

Notice that if a partial equivalence relation R is compatible with $\lambda_{\mathscr{C}}$ then by definition

$$t \simeq_{\lambda_{\mathscr{C}}} t' \quad \Longrightarrow \quad \overline{t}^R = \overline{t'}^R. \tag{4.2}$$

The family of $\lambda_{\mathscr{C}}-per$s can be provided with the usual semantic operators: arrow, intersection and product.

Definition 4.3.2 (*Arrow*)

If R and R' are two $\lambda_{\mathscr{C}}-per$s, then $R \to R'$ is given by

$$t = t' : R \to R' \quad \text{when} \quad \text{for any } u, u',\ u = u' : R \implies tu = t'u' : R'.$$

It is also a $\lambda_{\mathscr{C}}-per$.

The definition of the arrow in the $\lambda_{\mathscr{C}}-per$s is more fine than in many structures (such as reducibility candidates, *cf.* Def. 3.1.3). Indeed, for any t and any R, R', $t \in R \to R'$ implies $\forall u \in R,\ tu \in R'$. However this condition is conversly not sufficient to conclude that $t \in R \to R'$. Take for instance R the closure of $\{(\mathsf{c}_1, \mathsf{c}_1), (\mathsf{c}_2, \mathsf{c}_2)\}$ under $\lambda_{\mathscr{C}}$-equivalence, and let R' be $\simeq_{\lambda_{\mathscr{C}}}$. Then for any $u \in R$, $(\lambda x.x)u \in R'$ (*i.e.* $(\lambda x.x)u \simeq_{\lambda_{\mathscr{C}}} (\lambda x.x)u$). Nevertheless $\lambda x.x \notin R \to R'$, since $\mathsf{c}_1 = \mathsf{c}_2 : R$ and $(\lambda x.x)\mathsf{c}_1 \not\simeq_{\lambda_{\mathscr{C}}} (\lambda x.x)\mathsf{c}_2$.

The product on $\lambda_{\mathscr{C}}-per$s is defined using the usual Church's tuple encoding in the lambda calculus:

$$\begin{array}{rcll} (\!|x_1, \ldots, x_k|\!)_k & = & \lambda f.f\, x_1 \ldots x_k & \\ \pi_i^k & = & \lambda p.p\,(\lambda x_1 \ldots x_k.x_i) & (i \in [\![1..k]\!]) \end{array}$$

(We may write $(\!|x, y|\!)$ for $(\!|x, y|\!)_2$ and π_i for π_i^2). It satisfies the expected equivalence: $\pi_i^k\ (\!|t_1, \ldots, t_k|\!)_k \simeq_{\lambda_{\mathscr{C}}} t_i$.

Definition 4.3.3 (*Product*)

Given $(R_i)_{1 \le i \le k}$ a finite family of $\lambda_{\mathscr{C}}-per$s, the $\lambda_{\mathscr{C}}-per$ $R_1 \times \ldots \times R_k$ is defined by

$$t = u : R_1 \times \ldots \times R_k \qquad \text{when for each } i \in [\![1..k]\!], \quad \pi_i^k t = \pi_i^k u : R_i$$

Remark 4.3.1 . In $\lambda_{\mathscr{C}}-per$s, we get surjective pairing for free in any product $\lambda_{\mathscr{C}}-per$. Indeed, if $(\!|\pi_1 t, \pi_2 t|\!) \in A_1 \times A_2$ where A_1 and A_2 are two $\lambda_{\mathscr{C}}-per$s, it means (by definition of $A_1 \times A_2$) $\pi_i(\!|\pi_1 t, \pi_2 t|\!) \in A_i$. Since $\pi_i(\!|\pi_1 t\,,\, \pi_2 t|\!) \simeq_{\lambda_{\mathscr{C}}} \pi_i t$, then $\pi_i(\!|\pi_1 t\,,\, \pi_2 t|\!) = \pi_i t : A_i$. Thus $(\!|\pi_1 t\,,\, \pi_2 t|\!) = t : A_1 \times A_2$.

4.3.2 Category $\mathbb{P}\mathrm{er}_{\lambda_{\mathscr{C}}}$

The category $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$is defined as follows:

- an *object* is a partial equivalence relation compatible with $\simeq_{\lambda_{\mathscr{C}}}$.

- a *morphism* from A to B is an equivalence class in $\mathrm{dom}(A \to B)$.
  We may write $\overline{t} : A \to B$ instead of $\overline{t}^{A\to B} : A \to B$ when it raises no ambiguity.

- the *identity morphism* of a $\lambda_{\mathscr{C}}$*−per* A is $Id_A = \overline{\lambda x.x} \in \mathrm{dom}(A \to A)$.
  Indeed, if $t = t' : A$, then $(\lambda x.x)t \simeq_{\lambda_{\mathscr{C}}} t$ and $(\lambda x.x)t' \simeq_{\lambda_{\mathscr{C}}} t'$ and so $(\lambda x.x)t = (\lambda x.x)t' : A$ by compatibility with $\simeq_{\lambda_{\mathscr{C}}}$.

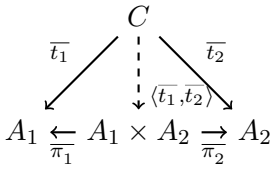
- the *composition* of $\overline{t} : A \to B$ and $\overline{t'} : B \to C$ is $\overline{t};\overline{t'} = \overline{\lambda z.t'(tz)}^{A\to C}$.
  It is well defined since $(\overline{t};\overline{t'}) \in \mathrm{dom}(A \to C)$. Indeed, for any $u = u' : A$, $(\lambda z.t'(tz))u$ reduces on $t'(tu)$. But $tu = tu' : B$, hence $t'(tu) = t'(tu') : C$ and $t'(tu') \simeq_{\lambda_{\mathscr{C}}} (\lambda z.t'(tz))u'$. Since C is compatible with $\simeq_{\lambda_{\mathscr{C}}}$ this entails $(\lambda z.t'(tz))u = (\lambda z.t'(tz))u' : C$ and then $\lambda z.t'(tz) \in A \to C$. In the same way, we can check that $(\overline{t};\overline{\lambda x.x}^{B\to B}) = (\overline{\lambda x.x}^{A\to A};\overline{t}) = \overline{t}$, and also that this definition of composition is associative.

This defines correctly the category $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$. We also provide it with the structure of a Cartesian closed category:

Product The product $A_1 \times A_2$ of two objects is their $\lambda_{\mathscr{C}}$*−per* product (Def. 4.3.3), and the i^{th} projection morphism (for $i \in \{1,2\}$) is $\overline{\pi_i}^{A_1\times A_2\to A_i}$.

Given $\overline{t_1} : C \to A_1$ and $\overline{t_2} : C \to A_2$, the pairing of $\overline{t_1}$ and $\overline{t_2}$ is

$$\langle \overline{t_1}, \overline{t_2} \rangle = \overline{\lambda x.(\!|t_1 x, t_2 x|\!)}^{C\to A_1\times A_2} .$$



The following proposition ensures that this pairing is well-defined (it does not depends on the representative of $\overline{t_1}$ and $\overline{t_2}$ that we choose), and that it defines effectively a product in the category $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$:

Proposition. *If* $t_i = t'_i : C \to A_i$ *(for* $i = 1, 2$*) then*
$$\lambda x.(\!|t_1 x, t_2 x|\!) = \lambda x.(\!|t'_1 x, t'_2 x|\!) : C \to A_1 \times A_2.$$
Moreover, $\langle \overline{t_1}, \overline{t_2} \rangle$ *is the unique morphism such that the diagram above commutes.*

Proof : *The equality* $\lambda x.(\!|t_1 x, t_2 x|\!) = \lambda x.(\!|t'_1 x, t'_2 x|\!) : C \to A_1 \times A_2$ *and the commutation of the diagram directly comes from the compatibility of* A_1 *and* A_2 *with* $\simeq_{\lambda_{\mathscr{C}}}$. *We prove the uniqueness of the pairing. Assume there is a morphism* $h = \overline{t_h} : C \to A_1 \times A_2$ *such that* $h;\pi_i = \overline{t_i}^{C\to A_i}$ *(for* $i = 1$ *and* $i = 2$*). This means* $\lambda z.\pi_i(t_h z) = t_i : C \to A_i$. *Hence for any* $u = u' : C$ *we can deduce the following equations one after the other using compatibility with* $\simeq_{\lambda_{\mathscr{C}}}$:

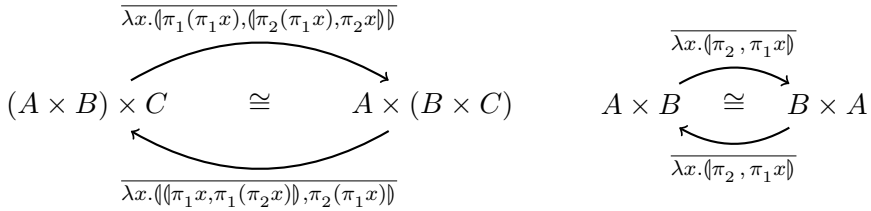
$$\begin{array}{rcll}
(\lambda z.\pi_i(t_h z))u & = & t_i\ u' & : A_i \\
\pi_i(t_h u) & = & t_i\ u' & : A_i \\
\pi_i(t_h u) & = & \pi_i(\!|t_1\ u', t_2\ u'|\!) & : A_i \\
t_h u & = & (\!|t_1 u', t_2 u'|\!) & : A_1 \times A_2 \\
t_h u & = & (\lambda x.(\!|t_1 x, t_2 x|\!))u' & : A_1 \times A_2 \\
t_h & = & \lambda x.(\!|t_1 x, t_2 x|\!) & : C \to A_1 \times A_2
\end{array}$$

Also $h = \langle t_1, t_2 \rangle$. □

By compatibility with the $\lambda_{\mathscr{C}}$-equivalence for any $t \in \mathrm{dom}(B \to C)$ and any $t' \in \mathrm{dom}(B' \to C')$, the product of $\overline{t}$ and $\overline{t'}$ is

$$\overline{t} \times \overline{t'} \quad = \quad \langle (\pi_1; \overline{t}), (\pi_2; \overline{t'}) \rangle \quad = \quad \overline{\lambda x.(\!|t(\pi_1 x), t'(\pi_2 x)|\!)} \tag{4.3}$$

We give the associativity and commutativity isomorphisms, that we may use in the following:



All those definitions are naturally extended to n-ary products.

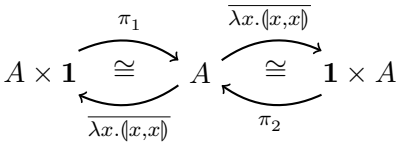
Remark 4.3.2 . There is an alternative definition for the product in $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$: given two $\lambda_{\mathscr{C}}$-pers A_1, A_2, define:

$$t = u : A_1 * A_2 \quad \text{when} \quad \begin{cases} t \simeq_{\lambda_{\mathscr{C}}} (\!|t_1, t_2|\!) \\ u \simeq_{\lambda_{\mathscr{C}}} (\!|u_1, u_2|\!) \end{cases} \quad \text{with } t_i = u_i : A_i \text{ (for } i = 1, 2).$$

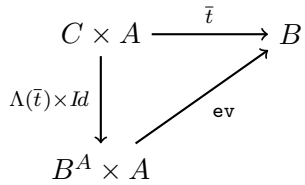
Together with projection morphisms $\overline{\pi_i}^{A_1 * A_2 \to A_i}$, this defines indeed a product in $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$.

Also $A_1 \times A_2 \cong A_1 * A_2$ by uniqueness of the product (Sec. 4.1.2). Furthermore, $t = u : A_1 * A_2$ implies $t = u : A_1 \times A_2$. The converse does not hold, and also the surjective pairing (Rem. 4.3.1) is not valid for $*$: let A_i be the closure of $\{(\pi_i \mathsf{c}, \pi_i \mathsf{c})\}$ under $\lambda_{\mathscr{C}}$-equivalence for $i = 1$ and $i = 2$. Then $(\!|\pi_1 \mathsf{c}, \pi_2 \mathsf{c}|\!) \in A_1 * A_2$ and so $(\!|\pi_1 \mathsf{c}, \pi_2 \mathsf{c}|\!) \in A_1 \times A_2$. By Rem. 4.3.1, $\mathsf{c} = (\!|\pi_1 \mathsf{c}, \pi_2 \mathsf{c}|\!) : A_1 \times A_2$. However there is no terms t_1, t_2 such that $\mathsf{c} \simeq_{\lambda_{\mathscr{C}}} (\!|t_1, t_2|\!)$, and thus $\mathsf{c} \neq (\!|\pi_1 \mathsf{c}, \pi_2 \mathsf{c}|\!) : A_1 * A_2$.

Terminal object The $\lambda_{\mathscr{C}}$*−per* $\mathbf{1} = \Lambda \times \Lambda$ is a terminal object in $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$. In fact, for any $\lambda_{\mathscr{C}}$*−per* A, the relation $A \to \mathbf{1}$ is equal to $\mathbf{1}$, and thus the unique equivalence class modulo $A \to \mathbf{1}$ is $!_A = \Lambda$.
This terminal object is indeed neutral for the product (up to the isomorphisms given in the diagram below).



Exponent Given two $\lambda_{\mathscr{C}}-pers$ A and B, B^A is the $\lambda_{\mathscr{C}}-per$ $A \to B$.



The curried form of a morphism $\bar{t} : C \times A \to B$ is

$$\Lambda(\bar{t}) = \overline{\lambda x.\lambda y.t\,(\!|x,y|\!)}^{C \to B^A}$$

and the morphism $\mathtt{ev}$ is $\overline{\lambda x.(\pi_1 x)(\pi_2 x)}$.

Again, $\Lambda(\ )$ is well-defined and together with $\mathtt{ev}$ it is indeed an exponent for A and B:

Proposition. *The definition of $\Lambda(\bar{t})$ does not depend on the representative of the class of t that we choose, and it is the unique morphism of $C \to B^A$ such that the diagram above commutes.*

Proof : *If $t = u : C \times A \to B$ then $\lambda x.\lambda y.t\,(\!|x,y|\!) = \lambda x.\lambda y.u\,(\!|x,y|\!) : C \to B^A$ by compatibility with $\simeq_{\lambda_{\mathscr{C}}}$. We show the commutation of the diagram:*

$$\begin{aligned}
\Lambda(\bar{t}) \times Id &= \overline{\lambda x.(\!|(\lambda x'.\lambda y.t\,(\!|x',y|\!))\ (\pi_1 x)\ ,\ \pi_2 x|\!)}^{C\times A \to B^A \times A} && (4.3)\\
&= \overline{\lambda x.(\!|\lambda y.t\,(\!|\pi_1 x, y|\!)\ ,\ \pi_2 x|\!)}^{(C\times A)\to(B^A \times C)}\\
(\Lambda(\bar{t}) \times Id);\mathtt{ev} &= \overline{\lambda z.(\lambda x.(\pi_1 x)(\pi_2 x))\ \big((\lambda x.(\!|\lambda y.t\,(\!|\pi_1 x, y|\!)\ ,\ \pi_2 x|\!))z\big)}^{C\times A \to B}\\
&= \overline{\lambda z.(\lambda x.(\pi_1 x)(\pi_2 x))\ (\!|\lambda y.t\,(\!|(\pi_1 z), y|\!)\ ,\ \pi_2 z|\!)}^{C\times A \to B}\\
&= \overline{\lambda z.t\,(\!|\pi_1 z, \pi_2 z|\!)}^{(C\times A)\to B}
\end{aligned}$$

Using Rem. 4.3.1, we conclude that $\lambda z.t\,(\!|\pi_1 z, \pi_2 z|\!) = t : (C \times A) \to B$. Moreover, $\Lambda(\bar{t})$ is the unique morphism that makes the diagram above commute (same method of proof as for the uniqueness of pairing).

Also those definitions provide actually a Cartesian closed category.

Proposition 4.3.1. *The category $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$ is a* CCC.

4.3.3 Syntactic model in $\mathbb{P}\mathrm{er}_{\lambda_{\mathscr{C}}}$.

The set of $\lambda_{\mathscr{C}}-pers$ forms a complete partial order for the inclusion order. The least element is the empty relation, the greater one is $\Lambda \times \Lambda$, and the least-upper bound of a family of $\lambda_{\mathscr{C}}-pers$ is the transitive closure of its union. Within these $\lambda_{\mathscr{C}}-pers$, some are *total*: each term is accessible by them. They are the equivalence relations on Λ compatible with $\simeq_{\lambda_{\mathscr{C}}}$. The smallest one is $\simeq_{\lambda_{\mathscr{C}}}$.

We call D the $\lambda_{\mathscr{C}}-per$ $\simeq_{\lambda_{\mathscr{C}}}$. We will see that it is a reflexive object in $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$, and use it to construct a $\lambda_{\mathscr{C}}$-model in $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$.

Lemma 4.3.2. *In $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$, $D = D^D$.*

Proof : **$\subseteq$:** *If $t = t' : D$, then $u = u' : D$ implies $tu = t'u' : D$ by definition of D. This means $t = t' : D^D$*

$\supseteq$: *Assume $t = t' : D^D$, and choose x not free in t nor t'. Since $x = x : D$, then $tx = t'x : D$. So $\lambda x.tx = \lambda x.t'x : D$ by contextual closure, and $t = t' : D$ by* LamApp. □

Definition 4.3.4 (*Syntactic model*)

The syntactic model (or *PER model*) of the lambda calculus with constructors is $\mathscr{M}_{synt} = (\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}, D, Id_D, Id_D, (c_i^*)_{1\leq i\leq n}, \mathtt{case}, \grave{}\,)$, where:

- D is the relation $\simeq_{\lambda_{\mathscr{C}}}$.
- for $\mathtt{c} \in \mathscr{C}$, $c^* = \overline{\lambda x.\mathtt{c}}^{\mathbf{1}\to D}$.
- $\mathtt{case} = \overline{\lambda x.\{\!|(\mathtt{c_i} \mapsto \pi_i^n(\pi_1 x))_{1\leq i\leq n}|\!\} \cdot \pi_2 x}^{(D^n\times D)\to D}$
- $\grave{} = \overline{\lambda x.\{\!|\ |\!\} \cdot \mathtt{c_1}}^{\mathbf{1}\to D}$

In fact we could have choosen any constructor $\mathtt{c}_i$ instead of $\mathtt{c}_1$ in the definition of $\grave{}$.

Proposition 4.3.3. *$\mathscr{M}_{synt}$ is a $\lambda_{\mathscr{C}}$-model.*

Proof : *We check every point of Def. 4.2.2.*

- $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$ *is a Cartesian closed category by Prop. 4.3.1.*
- *D is equal (and thus isomorphic) to D^D by Lem. 4.3.2.*
- *$c^* \in \mathrm{dom}(\mathbf{1}\to D)$ for each $\mathtt{c}\in\mathscr{C}$. Indeed, for any terms u, u', $(\lambda x.\mathtt{c})\ u \simeq_{\lambda_{\mathscr{C}}} \mathtt{c} \simeq_{\lambda_{\mathscr{C}}} (\lambda x.\mathtt{c})\ u'$. Hence $\lambda x.\mathtt{c} = \lambda x.\mathtt{c} : \mathbf{1}\to D$. In the same way, $\grave{} \in \mathrm{dom}(\mathbf{1}\to D)$.*
- *$\mathtt{app} = Id_D$ (and $\mathtt{lam} = Id_D$) is trivially a morphism of $D\to D^D$ (resp. of $D^D\to D$) by Lem. 4.3.2. We show that $\lambda x.\{\!|(\mathtt{c_i}\mapsto\pi_i^n(\pi_1 x))_{i=1}^n|\!\}\cdot\pi_2 x \in (D^n\times D)\to D$. Let $t = u : (D^n\times D)$. By definition, $\pi_i^n(\pi_1 t) = \pi_i^n(\pi_1 u) : D$, and $\pi_2 t = \pi_2 u : D$. Thus*
$$\begin{aligned}\big(\lambda x.\{\!|(\mathtt{c_i}\mapsto\pi_i^n(\pi_1 x))_{i=1}^n|\!\}\cdot\pi_2 x\big)t &\simeq_{\lambda_{\mathscr{C}}} \{\!|(\mathtt{c_i}\mapsto\pi_i^n(\pi_1 t))_{i=1}^n|\!\}\cdot\pi_2\ t\\ &\simeq_{\lambda_{\mathscr{C}}} \{\!|(\mathtt{c_i}\mapsto\pi_i^n(\pi_1 u))_{i=1}^n|\!\}\cdot\pi_2\ u\\ &\simeq_{\lambda_{\mathscr{C}}} \big(\lambda x.\{\!|(\mathtt{c_i}\mapsto\pi_i^n(\pi_1 x))_{i=1}^n|\!\}\cdot\pi_2 x\big)u\end{aligned}$$
- *By Prop. 4.2.1 the six diagrams of Fig. 4.1 commute since $(D1)$ trivially commutes for $\mathtt{lam} = \mathtt{app} = Id_D$, and diagrams $(D2)$, $(D3)$, $(D5)$ and $(D6)$ commute too (cf. appendix. A.1.2)* □

Remember that a term is *defined* if it has no blocking subterm $\{\!|\theta|\!\}\cdot\mathtt{c}$ where $\mathtt{c}\notin\mathrm{dom}(\theta)$, and it is *hereditarily defined* when all its reducts (including itself) are defined. Notice that $\lambda_{\mathscr{C}}$-models do not separate some undefined terms that are not $\lambda_{\mathscr{C}}$-equivalent. That is because the interpretation of a term first "completes" each case-binding with branches $\mathtt{c}_j\mapsto\grave{}$ if $\mathtt{c}_j$ is not in its domain (*cf.* the discussion page 58). So in the PER model, undefined terms are "unblocked" and the rule CaseCons can be performed (and give $\{\!|\ |\!\}\cdot\mathtt{c_1}$).

That is the reason why the completeness theorem (Theo. 4.2) is restricted to hereditarily defined terms. In this section we first formalise the idea of case-binding completion. This enables an explicit definition of the interpretation of a term in the PER model, so that we can prove the completeness theorem.

Definition 4.3.5 (*Case-completion*)

The case-completion of a term is defined by induction:

$$\widetilde{x} = x \qquad \widetilde{\lambda x.t} = \lambda x.\widetilde{t} \qquad \widetilde{\{\!|\theta|\!\}\cdot t} = \{\!|\widetilde{\theta}|\!\}\cdot\widetilde{t}$$
$$\widetilde{\mathsf{c}} = \widetilde{\mathsf{c}} \qquad \widetilde{tu} = \widetilde{t}\widetilde{u}$$
$$\widetilde{\theta} = \{\mathsf{c}_i \mapsto u'_i / 1 \le i \le n\} \qquad \text{with } u'_i = \begin{cases} \widetilde{u_i} & \text{if } \mathsf{c}_i \mapsto u_i \in \theta \\ \{\!|\,|\!\}\cdot \mathsf{c}_1 & \text{if } \mathsf{c}_i \notin \mathrm{dom}(\theta) \end{cases}$$

Fact 4.3.3 . This case-completion does not unify different defined terms: if two defined terms have the same case-completion, then they are equal.

Proposition 4.3.4. *In the model $\mathscr{M}_{synt}$, the interpretation of a term t in a context $\Gamma = x_1; \cdots ; x_k$ is*

$$[t]_\Gamma = \overline{\lambda x.\widetilde{t}[x_i := \pi_i^k x]}^{D^k \to D} \ ,$$

with x fresh in t.

Proof : *The proof proceeds by structural induction on t. If $t = x_i$ or $t = \mathsf{c}$, we just have to write the definition of $[t]_\Gamma$. If $t = \lambda x_{k+1}.t_0$ or $t = t_1 t_2$, the equation is straightforward from definition of $[t]_\Gamma$ and induction hypothesis. We detail the proof when $t = \{\!|\theta|\!\}\cdot u$:*
$[t]_\Gamma = \langle [\theta]_\Gamma; [u]_\Gamma\rangle; \mathsf{case}$, with $[\theta]_\Gamma = \langle f_1, \ldots, f_n\rangle$ where $f_j = [u_j]_\Gamma$ if $\mathsf{c}_j \mapsto u_j \in \theta$, and $f_j = !_{D^k}; \mathsf{`}$ $(= \overline{\lambda x.\{\!|\,|\!\}\cdot \mathsf{c}_1}^{D^k \to D})$ if $\mathsf{c}_j \notin \mathrm{dom}(\theta)$. So

$$[t]_\Gamma = \overline{\lambda x.t_{\mathsf{case}}\ (\!|t_\theta x, t_u x|\!)}^{D^k \to D}$$

with $\mathsf{case} = \overline{t_{\mathsf{case}}}^{D^n \times D \to D}$, $[\theta]_\Gamma = \overline{t_\theta}^{D^k \to D^n}$, and $[u]_\Gamma = \overline{t_u}^{D^k \to D}$. By induction hypothesis, we can chose $t_u = \lambda x.\widetilde{u}[x_i := \pi_i^k x]$, and $t_\theta = \lambda x.(\!|t_1 x, \ldots, t_n x|\!)_n$ with $t_j = \lambda x.\widetilde{u_j}[x_i := \pi_i^k x]$ if $\mathsf{c}_j \mapsto u_j \in \theta$, and $t_j = \lambda x.\{\!|\,|\!\}\cdot \mathsf{c}_1$ if $\mathsf{c}_j \notin \mathrm{dom}(\theta)$.
Also $\lambda x.t_{\mathsf{case}}\ (\!|t_\theta x, t_u x|\!) \simeq_{\lambda_{\mathscr{C}}} \lambda x.t_{\mathsf{case}}\ (\!|(\!|t_1 x, \ldots, t_n x|\!)_n\ \widetilde{u}[x_i := \pi_i^k x]\,|\!)$
$\simeq_{\lambda_{\mathscr{C}}} \lambda x.\widetilde{\{\!|(\mathsf{c}_j \mapsto t_j x)_{j=1}^n|\!\}\cdot u}[x_i := \pi_i^k x]$
$\simeq_{\lambda_{\mathscr{C}}} \lambda x.\widetilde{\{\!|\theta|\!\}\cdot u}\,[x_i := \pi_i^k]$
Indeed, $t_j x \simeq_{\lambda_{\mathscr{C}}} \widetilde{u_j}[x_i := \pi_i^k x]$ if $\mathsf{c}_j \mapsto u_j \in \theta$, and $t_j \simeq_{\lambda_{\mathscr{C}}} \{\!|\,|\!\}\cdot \mathsf{c}_1$ if $\mathsf{c}_j \notin \mathrm{dom}(\theta)$.
Since $D^k \to D$ is compatible with $\simeq_{\lambda_{\mathscr{C}}}$, $[t]_\Gamma = \overline{\lambda x.\widetilde{t}[x_i := \pi_i^k x]}^{D^k \to D}$. □

4.3.4 Completeness result.

As it is emphasised by Rem. 4.2.1, two match-failing terms can have the same interpretation even if they are not $\lambda_{\mathscr{C}}$-convertible. For instance,

$$[\{\!|\,|\!\}\cdot \mathsf{c}_2] = [\{\!|\mathsf{c}_1 \mapsto \lambda x.x|\!\}\cdot \mathsf{c}_2]_\Gamma = \mathsf{`} \ .$$

Now we show that this problem is specific to undefined terms, and that two hereditarily defined terms that are not $\lambda_{\mathscr{C}}$-equivalent are separated in the PER-model. In that view, we use the explicit definition of the interpretation of terms in the syntactic model (Prop. 4.3.4) and we also prove the following proposition:

Proposition 4.3.5. *Let t_1 and t_2 be two hereditarily defined terms. Then*

$$\widetilde{t_1} \simeq_{\lambda_{\mathscr{C}}} \widetilde{t_2} \quad \Longrightarrow \quad t_1 \simeq_{\lambda_{\mathscr{C}}} t_2$$

The proof of this proposition uses rewriting techniques, and relies on several lemmas (whose proofs are given in appendix A.1.3).

Fact 4.3.4 . The definition of case-completion (Def. 4.3.5) preserves all $\lambda_{\mathscr{C}}$-redexes. Also if $t \to u$ then $\widetilde{t} \to \widetilde{u}$, and if $\widetilde{t}$ is a normal form then so is t.

The two following lemmas establish to what extent the converse holds.

Lemma 4.3.6 ($\lambda^-_{\mathscr{C}}$-reduction on completed terms)**.** *Let t be a defined term. Then, for any term t',*

$$\widetilde{t} \to_{\lambda^-_{\mathscr{C}}} t' \quad \textit{implies} \quad t' = \widetilde{t_0} \ \textit{ for some } t_0 \textit{ such that } t \to t_0.$$

(Remember that $\lambda^-_{\mathscr{C}}$ denotes the $\lambda_{\mathscr{C}}$-calculus without the rule CaseCase).

Lemma 4.3.7 (CaseCase reduction on completed terms)**.** *For any terms t, t',*

$$\widetilde{t} \to_{cc} t' \quad \textit{implies} \quad t' \to^*_{cc} \widetilde{t_0} \ \textit{ for some } t_0 \textit{ such that } t \to_{cc} t_0$$

where $\to_{cc}$ denotes a reduction with rule CaseCase.

Also the rule CaseCase does not have the same behaviour as the other rule *w.r.t.* case-completion. In the following, we pay special attention to this rule.

Proposition 4.3.8. *The reduction rule $\to_{cc}$ forms a confluent and strongly normalising rewriting system.*

Proof : *The confluence has been proved in [AMR09, Theo. 1] (cf. Sec. 2.1.5). This reduction rule is also strongly normalising, since it makes the structural measure of terms (Def. 2.1.2) decrease.* □

In this paragraph, we write $\Downarrow t$ the normal form of a term t *w.r.t.* $\to_{cc}$. It is characterised by the following equations:

$$\begin{array}{rcl@{\qquad}rcl}
\Downarrow x & = & x & \Downarrow \{\mathsf{c}_i \mapsto u_i \,/\, i\in I\} & = & \{\mathsf{c}_i \mapsto \Downarrow u_i \,/\, i\in I\} \\
\Downarrow \mathsf{c} & = & \mathsf{c} & \multicolumn{3}{l}{\text{If } t = x \mid \mathsf{c} \mid \lambda x.u \mid t_1 t_2 \text{ , then}} \\
\Downarrow \lambda x.t & = & \lambda x. \Downarrow t & \Downarrow \{\!|\theta|\!\} \cdot t & = & \{\!|\Downarrow \theta|\!\} \cdot \Downarrow t \\
\Downarrow (tu) & = & \Downarrow t \Downarrow u & \Downarrow \left(\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t\right) & = & \Downarrow \left(\{\!|\theta \circ \phi|\!\} \cdot t\right)
\end{array}$$

Lemma 4.3.9. *Commutation case-completion/*cc*-normal form*
For any term t,

$$\Downarrow (\widetilde{t}) = \widetilde{\Downarrow t} \ .$$

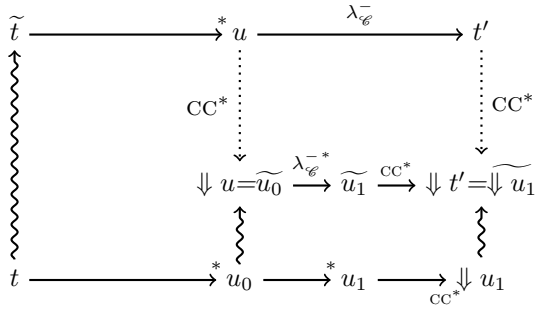
Lemma 4.3.10. *For any terms t, t', if $t \to_{\lambda^-_{\mathscr{C}}} t'$ then there exists a term u such that*

$$\Downarrow t \ \to^*_{\lambda^-_{\mathscr{C}}} \ u \ \to^*_{cc} \Downarrow t' \ .$$

Corollary 4.3.11. *If t is hereditarily defined, then for any t',*

$$\widetilde{t} \to^* t' \quad \textit{implies} \quad \Downarrow t' = \widetilde{t_0} \textit{ for some } t_0 \textit{ such that } t \to^* t_0 \ .$$

PROOF : *By induction on the reduction $\widetilde{t} \to^* t'$.*
If $\widetilde{t} = t'$, take $t_0 = t$. Now assume $\widetilde{t} \to^* u \to_R t'$. By induction hypothesis, there is some u_0 such that $\Downarrow u = \widetilde{u_0}$ and $t \to^* u_0$. If u reduces on t' with the rule $R =$ CASECASE, then $\Downarrow t' = \Downarrow u = \widetilde{u_0}$, and $t_0 = u_0$ does the job. Otherwise, $\widetilde{t} \to^* u \to_{\lambda^-_{\mathscr{C}}} t'$.

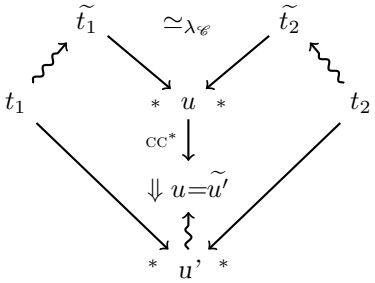


First of all, $u \to_{\lambda^-_{\mathscr{C}}} t'$ implies $\Downarrow u \ \to^*_{\lambda^-_{\mathscr{C}}} \ u' \ \to^*_{cc} \Downarrow t'$ for some u' (Lem. 4.3.10). Also $\widetilde{u_0} \ \to^*_{\lambda^-_{\mathscr{C}}} \ u'$, and thus $u' = \widetilde{u_1}$ for some term u_1 such that $u_0 \to^*_{\lambda^-_{\mathscr{C}}} u_1$ (Lem. 4.3.6, since u_0 is defined). Moreover, $\widetilde{u_1} \to^*_{cc} \Downarrow t'$ implies that $\Downarrow t'$ is the CASECASE normal form of $\widetilde{u_1}$. Hence $\Downarrow t' \ = \Downarrow \widetilde{u_1} \ = \ \widetilde{\Downarrow u_1}$ (by Lem. 4.3.9). Also we can chose $t_0 = \Downarrow u_1$. □

Now we have all the ingredients we need to prove that the case-completion preserves the $\lambda_{\mathscr{C}}$-equivalence on hereditarily defined terms.

PROOF : *(of Prop. 4.3.5).*

Let t_1, t_2 hereditarily defined such that $\widetilde{t_1} \simeq_{\lambda_{\mathscr{C}}} \widetilde{t_2}$. Since the $\lambda_{\mathscr{C}}$-calculus satisfies the Church-Rösser property, there is a term u such that $\widetilde{t_1} \to^* u$ and $\widetilde{t_2} \to^* u$.
Hence Cor. 4.3.11 provides a term u' such that $\Downarrow u = \widetilde{u'}$, and $t_i \to^* u'$ for each $i \in \{1, 2\}$. Thus $t_1 \simeq_{\lambda_{\mathscr{C}}} u' \simeq_{\lambda_{\mathscr{C}}} t_2$.



□

Together with the explicit definition of the interpretation of a term in the PER-model, this gives the result of completeness of $\lambda_{\mathscr{C}}$-models for terms with no match failure.

Corollary 4.3.12 (Completeness)**.** *Let t_1 and t_2 be two hereditarily defined terms whose free variables are in $\Gamma = \{x_1, \ldots, x_k\}$ such that $[t_1]_\Gamma = [t_2]_\Gamma$ in the syntactic model $\mathscr{M}_{synt}$, then $t_1 \simeq_{\lambda_{\mathscr{C}}} t_2$.*

PROOF : *By Prop. 4.3.4, if t_1 and t_2 have the same interpretation in $\mathscr{M}_{synt}$, it means that*

$$\overline{\lambda x.\widetilde{t_1}[x_i := \pi_i^k x]}^{D^k \to D} = \overline{\lambda x.\widetilde{t_2}[x_i := \pi_i^k x]}^{D^k \to D} .$$

Hence $(\lambda x.\widetilde{t_1}[x_i := \pi_i^k x])\ (\!|x_1, \ldots, x_k|\!)_k = (\lambda x.\widetilde{t_2}[x_i := \pi_i^k x])\ (\!|x_1, \ldots, x_k|\!)_k : D$. Since D is the $\lambda_{\mathscr{C}}$-equivalence relation on terms, it means that $\widetilde{t_1} \simeq_{\lambda_{\mathscr{C}}} \widetilde{t_2}$, which entails $t_1 \simeq_{\lambda_{\mathscr{C}}} t_2$ by Prop. 4.3.5. □

A fortiori if two hereditarily defined terms have the same interpretation in *any* $\lambda_{\mathscr{C}}$-model then they are $\lambda_{\mathscr{C}}$-equivalent, since $\mathscr{M}_{synt}$ is a $\lambda_{\mathscr{C}}$-model. This achieves the proof of Completeness theorem (Theo. 4.2).

Conclusion

The PER model does not separate every terms that are not $\lambda_{\mathscr{C}}$-equivalent (*cf.* Rem. 4.2.1), but it always assigns different denotations to two terms that are separated by the separation theorem (Sec. 2.1.5) [2].

The restriction of the completeness theorem to terms with no match failure is due to the manner in which we interpret the case-bindings. Since the denotation we give to them is a point of D^n, it requires to "fill" artificially every undefined branch of a case-binding.

A way to cope with this problem could be to first identify the domain $I \subseteq [\![1..n]\!]$ of a case-binding $\theta = \{\mathsf{c}_i \mapsto u_i / i \in I\}$, and interpret it by the point $(u_i)_{i \in I}$ of D^{n_I} (where n_I is the cardinal of I). The object that represents case-bindings would then be the sum (the dual notion of product) $\sum_{I \subseteq [\![1..n]\!]} D^{n_I}$. However, some difficulties arise to define the case composition. Also a good notion of $\lambda_{\mathscr{C}}$-model that separates *any* non-equivalent terms, even the match failure, remains to be defined.

[2] The syntactic model actually separates more terms than the separation property. For instance, terms $\{\!|\ |\!\} \cdot \mathsf{c}$ and $\{\!|\ |\!\} \cdot \mathsf{c}u$ are not separable by any context, yet they do not receive the same denotation in the PER model.

Chapter 5

CPS and Classical model

Continuation passing style (CPS) is a programming paradigm in which the control flow (the *continuation*) is manipulated by the program as an explicit parameter. The CPS translation of the terms of a given language consists in making explicit the evaluation context. Also translating terms in continuation passing style amounts to describe their evaluation in an abstract machine [Plo75].

In the study of programming languages, these translations are mostly used to translate a language with side effects and control into a purely functional language [SF92], or to simulate a reduction strategy into an other one [Fis93, OLT94]. Furthermore a new class of denotational models, the continuation semantics, arises from these translations [Mog91]. Concerning Logic, Lafont [Laf91] has shown that it corresponds to a $\neg\neg$-translation of classical logic into (a fragment of) intuitionistic logic.

In this chapter, we propose a CPS translation of the (complete) $\lambda_{\mathscr{C}}$-calculus into the lambda calculus with pairs. Therefore, we first describe an abstract machine (inspired by the Krivine's abstract machine) for the lambda calculus with constructors. Then we show that the resulting CPS is a correct simulation. Finally, we set up a notion of a continuation model for the $\lambda_{\mathscr{C}}$-calculus, and we show that it actually defines $\lambda_{\mathscr{C}}$-models in the sense of the previous chapter. This gives rise to some non syntactic models for the lambda calculus with constructors.

5.1 $\lambda_{\mathscr{C}}$-calculus and stack machines

5.1.1 Abstract machines and commutation rules

A stack-based abstract machine represents the execution of a program (a closed term t) in a context (a stack π). Also an execution state is represented by a pair

$$t \quad \star \quad \pi \ .$$

The evaluation context consists of value eliminators, and an evaluation step describes the interaction between a value in head position of the program and the eliminator at the top of the stack.

In the pure lambda-calculus, there is only one kind of values: functions. They are constructed with a lambda-abstraction, and deconstructed by applying an argument. When a lambda-abstraction "meets" an argument, a β-reduction is performed:

$$\lambda x.t \;\star\; u \cdot \pi \quad \blacktriangleright \quad t[x := u] \;\star\; \pi \qquad (\texttt{Pop})$$

When the evaluated program is not a lambda-abstraction it is necessarily an application. The argument is then pushed on the stack:

$$tu \;\star\; \pi \quad \blacktriangleright \quad t \;\star\; u \cdot \pi \qquad (\texttt{Push})$$

With these two evaluation rules, the execution process exactly simulates the weak head reduction evaluation. The process stops when a lambda-abstraction appears in front of an empty stack: $\lambda x.t \;\star\; [\,]$.
This is the only blocked configuration if the initial term was closed. We consider then that the program has been executed properly and returns a value ($\lambda x.t$).

Problems arise when the program uses different kinds of values. In the syntax of lambda-calculus with constructors, functions coexist with data structures, each coming with its construction method and its eliminator:

	Construction	Elimination
Functions	lambda-abstraction	application
Data-structures	constructors	pattern-matching

In an abstract machine for the $\lambda_{\mathscr{C}}$-calculus, the stack contains arguments (coming from applications) but also case-bindings (coming from case-constructs). In addition to the `Pop` and `Push` rules for functions, there are similar rules for constructed values:

$$\mathsf{c} \;\star\; \theta \cdot \pi \quad \blacktriangleright \quad \theta_c \;\star\; \pi \qquad (\texttt{Pop}_{\mathsf{c}})$$
$$\{\!|\theta|\!\} \cdot t \;\star\; \pi \quad \blacktriangleright \quad t \;\star\; \theta \cdot \pi \qquad (\texttt{Push}_{\mathsf{c}})$$

In most programming languages with case-analysis, those would be the only rules (concerning functions and constructed values). In that setting, three blocked situations can occur although the stack is not empty:

$$\begin{array}{rl} \mathsf{c} \;\star\; \theta \cdot \pi & (\text{with } \mathsf{c} \notin \mathrm{dom}(\theta)) \\ \mathsf{c} \;\star\; u \cdot \pi & \\ \lambda x.t \;\star\; \theta \cdot \pi & \end{array}$$

The first case corresponds to a match-failure. In the lambda-calculus with constructors we do not try to handle them, also we let this configuration as blocked.

The last two cases do not necessarily correspond to a blocked execution in the lambda calculus with constructors. Indeed, the rule CaseApp enables the commutation between an argument and a case-binding in the stack:

$$\begin{array}{ccc} t \star u \cdot \theta \cdot \pi & \blacktriangleright & t \star \theta \cdot u \cdot \pi \\ \{\!|\theta|\!\} \cdot (tu) & \multimap & (\{\!|\theta|\!\} \cdot t)u \end{array}$$

Remark 5.1.1 . Whereas the rule CaseApp exactly corresponds to the stack transformation $u \cdot \theta \cdot \pi \hookrightarrow \theta \cdot u \cdot \pi$, there is no such a rule for the transformation $\theta \cdot u \cdot \pi \hookrightarrow u \cdot \theta \cdot \pi$. However, it can be seen as a consequence of the rule CaseLam. Indeed this rule allows the following execution:

$$\begin{array}{rcl} \lambda x.t \star \theta \cdot u \cdot \pi & \simeq & \{\!|\theta|\!\} \cdot (\lambda x.t) \star u \cdot \pi \\ & \blacktriangleright & \lambda x.\{\!|\theta|\!\} \cdot t \star u \cdot \pi \\ & \blacktriangleright & \{\!|\theta|\!\} \cdot t[x := u] \star \pi \\ & \blacktriangleright & t[x := u] \star \theta \cdot \pi \ . \end{array}$$

The same behaviour is obtained by

$$\lambda x.t \star \theta \cdot u \cdot \pi \blacktriangleright \lambda x.t \star u \cdot \theta \cdot \pi \blacktriangleright t[x := u] \star \theta \cdot \pi \ . \qquad \square$$

Thus when a blocked configuration arises (a constructor with an argument at the top of the stack, or a λ-abstraction with a case-binding at the top of the stack), an abstract machine for $\lambda_{\mathscr{C}}$-calculus should "swap" adjacent arguments and case-bindings in the stack until a case-binding (*resp.* an argument) reaches the top position. Rule $\mathtt{Pop_c}$ (*resp.* $\mathtt{Pop}$) is then possible. Also we can consider the stacks *modulo* these commutations:

$$\begin{array}{rcl} \theta_1 \cdot u_1 \cdot u_2 \cdot \theta_2 \cdot u_3 & \simeq & \theta_1 \cdot \theta_2 \cdot u_1 \cdot u_2 \cdot u_3 \\ & \simeq & u_1 \cdot u_2 \cdot u_3 \cdot \theta_1 \cdot \theta_2 \end{array}$$

Such a swapping between adjacent eliminators of different kinds amounts to split the elimination context into two different stacks: one for the case-bindings and one for the arguments. A configuration of the machine is then written

$$\theta_1 \cdot \theta_2 \cdots \quad \star \quad t \quad \star \quad u_1 \cdot u_2 \cdots$$

Case context: a stack vs a case-bonding. The only reduction rules of the $\lambda_{\mathscr{C}}$-calculus (other than the daimon rules) that we did not consider so far in the design of the machine are the η-reduction and the case composition. These two rules have a similar role in the calculus: they are computationally useless, but they are necessary for the separation property (Sec. 2.1.5). Also we can see the rule CaseCase as the "η-reduction for case constructs":

$$\begin{array}{rcl} \lambda x.tx & \to & t \qquad x \notin \mathrm{fv}(t) \\ \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot u & \to & \{\!|\theta \circ \phi|\!\} \cdot u \end{array}$$

Both terms $\lambda x.tx$ and t reduce on the same term if we apply them to an argument (*i.e.* if t is seen as a function). In the same way, $\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot u$ and $\{\!|\theta \circ \phi|\!\} \cdot u$ have the same reduct if u is a data-structure (matchable by ϕ).

In the stack abstract machine for the pure lambda calculus, the η-reduction is not simulated. Nevertheless, we will not adopt the same choice concerning the rule CaseCase, for some reasons explained in Rem. 5.1.4.

Hence the rule $\texttt{Push}_\texttt{c}$, that would be, in a two stacks abstract machine,

$$\pi_c \cdot \theta \quad \star \quad \{\!|\phi|\!\} \cdot t \quad \star \quad \pi \qquad \blacktriangleright \qquad \pi_c \cdot \theta \cdot \phi \quad \star \quad t \quad \star \quad \pi$$

will be replced by

$$\pi_c \cdot \theta \quad \star \quad \{\!|\phi|\!\} \cdot t \quad \star \quad \pi \qquad \blacktriangleright \qquad \pi_c \cdot \theta \circ \phi \quad \star \quad t \quad \star \quad \pi$$

In fact the whole stack of case-bindings $\theta_1 \cdots \theta_k$ will be collapsed in only one case-binding $\theta_1 \circ \cdots \circ \theta_k$.

5.1.2 Stack abstract machine for $\lambda_{\mathscr{C}}$

The title of this section is cheating: the design we chose for the $\lambda_{\mathscr{C}}$-abstract machine (described in Fig. 5.1) uses a term, a stack of arguments and only one (optional) case-binding. Indeed, the rule CaseCase enables wrapping a whole (non empty) stack of case-bindings in only one case-binding. We justify this choice at the end of Sec. 5.1.3.

Processes:

Stacks:	π	$:=$	$\diamond \mid t \cdot \pi$
Optional cases:	$\langle\theta\rangle$	$:=$	$\diamond \mid \theta$
(State of) processes:	s	$:=$	$\langle\theta\rangle \star t \star \pi$

Execution rules:

$$\begin{array}{llllllllll}
\langle\theta\rangle & \star & \lambda x.t & \star & u \cdot \pi & \blacktriangleright & \langle\theta\rangle & \star & t[x := u] & \star & \pi & (\texttt{Pop}) \\
\langle\theta\rangle & \star & tu & \star & \pi & \blacktriangleright & \langle\theta\rangle & \star & t & \star & t \cdot \pi & (\texttt{Push}) \\
\\
\theta & \star & \texttt{c} & \star & \pi & \blacktriangleright & \diamond & \star & \theta_\texttt{c} & \star & \pi & (\texttt{Pop}_\texttt{c}) \\
\langle\theta\rangle & \star & \{\!|\phi|\!\} \cdot t & \star & \pi & \blacktriangleright & \langle\theta\rangle \cdot \phi & \star & t & \star & \pi & (\texttt{Push}_\texttt{c}) \\
\multicolumn{11}{c}{(\text{where } \diamond \cdot \phi = \phi \text{ and } \theta \cdot \phi = \theta \circ \phi.)} \\
\theta & \star & \maltese & \star & \pi & \blacktriangleright & \diamond & \star & \maltese & \star & \diamond & (\texttt{Exit})
\end{array}$$

Figure 5.1: $\lambda_{\mathscr{C}}$-abstract machine

A state s is *final* when there is no state s' such that $s \blacktriangleright s'$. Executing a program consists in *loading* its term, *running* the machine until a final state arises (if it does), and then *unloading*. Final states are of three different kinds:

- An error has occurred if a state $\langle\theta\rangle \star x \star \pi$ (open term) or $\theta \star \mathsf{c} \star \pi$ (with $\mathsf{c} \notin \mathrm{dom}(\theta)$, match failure) appears.
- Execution returns a value when one of these states appears: $\langle\theta\rangle \star \lambda x.t \star \diamond$ (function) or $\diamond \star \mathsf{c} \star \pi$ (data-structure).
- Program was exited if its evaluation leads to $\diamond \star \Lightning \star \diamond$.

Definition 5.1.1 (*Term evaluation*)

The function `load` maps every term t to the process $\diamond \star t \star \diamond$. The function `unload` is defined from final states to terms as follows:

$$
\begin{array}{lll}
\diamond \star x \star u_1 \cdots u_k \cdot \diamond & \xrightarrow{\texttt{unload}} & x u_1 \dots u_n \\
\theta \star x \star u_1 \cdots u_k \cdot \diamond & \xrightarrow{\texttt{unload}} & (\{\!|\theta|\!\} \cdot x) u_1 \dots u_n \\
\theta \star \mathsf{c} \star u_1 \cdots u_k \cdot \diamond \ (\mathsf{c} \notin \mathrm{dom}(\theta)) & \xrightarrow{\texttt{unload}} & (\{\!|\theta|\!\} \cdot \mathsf{c}) u_1 \dots u_n \\
\diamond \star \lambda x.t \star \diamond & \xrightarrow{\texttt{unload}} & \lambda x.t \\
\theta \star \lambda x.t \star \diamond \quad (x \notin \mathrm{fv}(\theta)) & \xrightarrow{\texttt{unload}} & \lambda x.(\{\!|\theta|\!\} \cdot t) \\
\diamond \star \mathsf{c} \star u_1 \cdots u_k \cdot \diamond & \xrightarrow{\texttt{unload}} & \mathsf{c} u_1 \dots u_n \\
\diamond \star \Lightning \star \diamond & \xrightarrow{\texttt{unload}} & \Lightning
\end{array}
$$

The evaluation function is then partially defined on terms by

$$eval(t) = \texttt{unload}(s) \qquad \text{if} \quad \texttt{load}(t) \ \blacktriangleright^* \ s, \quad s \text{ final.}$$

If such a state s exists it is unique, as the machine is deterministic. In this case $eval(t)$ is the weak head normal form of t (this is formalised in next section).

Example 5.1.2 (Evaluation of the predecessor.) . Remember that the predecessor function is

$$\texttt{pred} \ = \ \lambda x.\{\!|\theta_p|\!\} \cdot x \ , \qquad \text{with} \ \ \theta_p \ = \ \{\mathsf{0} \mapsto \mathsf{0}; \mathsf{S} \mapsto \lambda y.y\}$$

Write $\mathtt{2} = \mathsf{S}(\mathsf{S}\,\mathsf{0})$ and $\mathtt{3} = \mathsf{S}\,\mathtt{2}$ the unary integers 2 and 3. Then `pred 3` is evaluated as follows:

$$
\begin{array}{rcccccc}
\texttt{load(pred 3)} & = & \diamond & \star & \texttt{pred 3} & \star & \diamond \\
& \blacktriangleright & \diamond & \star & \texttt{pred} & \star & \mathtt{3} \\
& \blacktriangleright & \diamond & \star & \{\!|\theta_p|\!\} \cdot \mathtt{3} & \star & \diamond \\
& \blacktriangleright & \theta_p & \star & \mathtt{3} & \star & \diamond \\
& \blacktriangleright & \theta_p & \star & \mathsf{S} & \star & \mathtt{2} \\
& \blacktriangleright & \diamond & \star & \lambda y.y & \star & \mathtt{2} \\
& \blacktriangleright & \diamond & \star & \mathtt{2} & \star & \diamond
\end{array}
$$

Also $eval(\texttt{pred 3})$ is 2 as expected.

5.1.3 Weak head reduction

The $\lambda_{\mathscr{C}}$-abstract machine executes a (*call-by-name*) *weak head* reduction strategy on terms. We write $\rightarrow_{\mathrm{W}}$ the corresponding reduction for the $\lambda_{\mathscr{C}}$-calculus, and it is defined inductively by the following rules:

$$(\lambda x.t)u \rightarrow_{\mathrm{W}} t[x := u] \qquad \{\!|\theta|\!\} \cdot \lambda x.t \rightarrow_{\mathrm{W}} \lambda x.\{\!|\theta|\!\} \cdot t \qquad \{\!|\theta|\!\} \cdot \mathsf{c} \rightarrow_{\mathrm{W}} u \ \text{(if } \mathsf{c} \mapsto u \in \theta\text{)}$$

$$✠u \rightarrow_{\mathrm{W}} ✠ \qquad \{\!|\theta|\!\} \cdot tu \rightarrow_{\mathrm{W}} (\{\!|\theta|\!\} \cdot t)u \qquad \{\!|\theta|\!\} \cdot ✠ \rightarrow_{\mathrm{W}} ✠$$

$$\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t \rightarrow_{\mathrm{W}} \{\!|\theta \circ \phi|\!\} \cdot t \qquad \frac{t \rightarrow_{\mathrm{W}} t'}{tu \rightarrow_{\mathrm{W}} t'u}$$

Example 5.1.3 . Within this strategy, the reduction of a term $\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot (\lambda x.t)u$ is the following:

$$\begin{array}{lll} \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot (\lambda x.t)u & \rightarrow_{\mathrm{W}} & \{\!|\theta \circ \phi|\!\} \cdot (\lambda x.t)u \\ & \rightarrow_{\mathrm{W}} & (\{\!|\theta \circ \phi|\!\} \cdot \lambda x.t)\ u \\ & \rightarrow_{\mathrm{W}} & (\lambda x.\{\!|\theta \circ \phi|\!\} \cdot t)\ u \\ & \rightarrow_{\mathrm{W}} & \{\!|\theta \circ \phi|\!\} \cdot (t[x := u]) \end{array}$$

Normal form for this reduction are called *head normal form*. They correspond to the final states of the machine:

$$h \ := \lambda x.t \ \mid \ \mathsf{c}\vec{u} \ \mid \ ✠ \ \mid \ x\vec{u} \ \mid \ (\{\!|\theta|\!\} \cdot x)\vec{u} \ \mid \ (\{\!|\theta|\!\} \cdot \mathsf{c})\vec{u} \quad (\mathsf{c} \notin \mathrm{dom}(\theta))$$

The CPS-translation we define in the next part is based on the stack-abstract machine. Reduction $\rightarrow_{\mathrm{W}}$ just paraphrases the machine's execution, as established by the following proposition. However, we consider both: abstract machine for the intuitive ideas, and weak-head reduction for the formalism.

Proposition 5.1.1. *If a $\lambda_{\mathscr{C}}$-term t has a head normal form h, then $eval(t) = h$.*

Proof : *See Appendix A.2.1.* □

Remark 5.1.4 . Replacing a stack of case-bindings by only one case-binding in the stack machine remains to treat differently the eliminators of functions and the eliminators of data-structures (*cf.* the discussion p. 77). There was an alternative choice: to consider an abstract machine with to stacks, that would have executed a variant of the weak head reduction $\rightarrow_{\mathrm{W}}$ (let us write it $\Rightarrow_w$). Roughly speaking, this variant consists in replacing the rule $\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t \rightarrow_{\mathrm{W}} \{\!|\theta \circ \phi|\!\} \cdot t$ by

$$\frac{t \ \Rightarrow_w \ t'}{\{\!|\theta|\!\} \cdot t \Rightarrow_w \{\!|\theta|\!\} \cdot t'}$$

This last rule is, for the evaluation context of data-structures, the equivalent of the following rule for the functional context:

$$\frac{t \rightarrow_{\mathrm{W}} t'}{tu \rightarrow_{\mathrm{W}} t'u}$$

It amounts to say that the head sub-term of a term $\lambda\vec{x}.\big(\{\!|\theta_1|\!\}\cdot\cdots\cdot\{\!|\theta_k|\!\}\cdot h\big)\vec{u}$ (where h is neither an application nor a case construct) is h. On the other hand, the weak startegy $\rightarrowtriangle_{\mathrm{W}}$ that we defined behaves as if the head sub-term of such a term was $\{\!|\theta_1|\!\}\cdot\cdots\cdot\{\!|\theta_k|\!\}\cdot h$.

Moreover, with the alternative weak strategy $\Rightarrow_w$, two reduction rules remain not performed: the η-reduction and the case-composition. This would be coherent with the intuition that CaseCase is the η-reduction for case constructs (*cf.* p. 77), whereas the reduction $\rightarrowtriangle_{\mathrm{W}}$ that we consider here performs CaseCase reductions but no LamApp reduction.

However, the alternative weak strategy $\Rightarrow_w$ raises some technical difficulties. The first one is that it would require some precedence rules for beeing a strategy, otherwise the reduction is not deterministic: $\{\!|\theta|\!\}\cdot(\lambda x.t)u \Rightarrow_w \{\!|\theta|\!\}\cdot(t[x := u])$ and $\{\!|\theta|\!\}\cdot(\lambda x.t)u \Rightarrow_w (\{\!|\theta|\!\}\cdot\lambda x.t)\ u$.

Then, the abstract machine (with one case-binding and one stack) simulating the weak strategy $\rightarrowtriangle_{\mathrm{W}}$ will be used in the next section, to describe a CPS translation of the $\lambda_{\mathscr{C}}$-calculus into the lambda calculus with pairs. We want this translation to simulate *every* reduction of the lambda calculus with constructors, including LamApp and CaseCase. The rule LamApp will be simulated by the η-reduction of the lambda-calculus with pair, but there is no rule in this target calculus to simulate the case-composition. Also the case-bindings are composed *during* the translation. That is why we chose a stack machine that performs the rule CaseCase, and that has thereby a stack of arguments and only one case-binding for evaluation context. ∎

5.2 CPS translation

In this section and the following one we consider that the set of constructors is finite:

$$\mathscr{C} \;=\; \{\mathsf{c}_1,\ldots,\mathsf{c}_n\}\ .$$

We define a translation of the $\lambda_{\mathscr{C}}$-calculus into the lambda calculus with no constructors (and no case constructs), in the spirit of Plotkin's *Continuation Passing Style* (CPS) translation [Plo75]. The key difference is that continuations (representing the evaluation context of the abstract machine) are composed of a case-binding and a stack of arguments (and not only a stack):

$$\mathit{Continuations} \;=\; \mathit{Case\text{-}bindings} \times \mathit{Stacks}\ .$$

A $\lambda_{\mathscr{C}}$-term t is then translated by a lambda-term t^* that takes a continuation as argument, and then runs the abstract machine:

$$t^* \;=\; \lambda y.\mathit{let}\ \langle x_\theta, x_\pi\rangle := y\ \mathit{in}\ |x_\theta \star t \star x_\pi|\ .$$

This translation relies on the inductive definition of

$$|M_\theta \star t \star M_\pi|$$

where:

- M_θ is the translation of the case-binding θ of the evaluation context.
- M_π is the translation of the stack of arguments π of the evaluation context.
- t is the $\lambda_{\mathscr{C}}$-term that is beeing evaluated
- $|M_\theta \star t \star M_\pi|$ is the translation of the result of the evaluation of $\theta \star t \star \pi$.

This will be formalised in Sec. 5.2.2.

We chose the simply typed lambda calculus with pairs (*cf.* appendix B) as the target calculus. Indeed, a stack of terms will be implemented by overlapping pairs:

$$t_1 \cdot t_2 \cdot t_3 \cdots \quad \rightsquigarrow \quad \langle\!|t_1^*, \langle\!|t_2^*, \langle\!|t_3^*, \langle\!|\cdot, \cdots |\!\rangle|\!\rangle|\!\rangle$$

A case-binding θ is represented (as in the categorical models, *cf.* p. 58) by a n-tuple of terms (encoded with pairs in the usual way, *cf.* p. 114):

$$\theta \quad \rightsquigarrow \quad \langle\!|M_1, \ldots, M_n|\!\rangle_n$$

where M_i represents θ_{c_i} if $\mathsf{c}_i \in \mathrm{dom}(\theta)$, and is a special term representing match failure otherwise (it is detailed in Sec. 5.2.2).

5.2.1 The target calculus

In appendix B.1 we define λ_P, the simply typed lambda-calculus with pairs parametrised by a set of constant terms and a set of constant types, with some relations between them.

To translate the untyped $\lambda_{\mathscr{C}}$-calculus, we assume that the λ_P-calculus includes the following:

- a type C for the continuations, a type S for the stacks, and a type R corresponding to (the encoding of) the results returned by the evaluation of the stack machine.
  Then the translation of a term is a function from continuations to results. Also their type is $D = C \to R$. We also write D^n the n-ary product of D.
- two terms $\downarrow_s$ and $\uparrow_s$ (sometimes called "`cons`" and "`uncons`" in programming languages) for the implementation of a stack by overlapping pairs of (translated) terms. They must satisfy the flolowing typing and rewriting rules:
$$\begin{array}{ll} \vdash_p \uparrow_s : & S \to (D \times S) \\ \vdash_p \downarrow_s : & (D \times S) \to S \end{array} \qquad \begin{array}{ll} \downarrow_s (\uparrow_s M) & \twoheadrightarrow_p M \\ \uparrow_s (\downarrow_s M) & \twoheadrightarrow_p M \end{array}$$
- two terms to "unfold" a continuation into a pair of case-binding and a stack, and conversely, that satisfy the following rules:
$$\begin{array}{ll} \vdash_p \uparrow_c : & C \to (D^n \times S) \\ \vdash_p \downarrow_c : & (D^n \times S) \to C \end{array} \qquad \begin{array}{ll} \downarrow_c (\uparrow_c M) & \twoheadrightarrow_p M \\ \uparrow_c (\downarrow_c M) & \twoheadrightarrow_p M \end{array}$$

Notice that this defines a well-designed parameter (Def. B.1.1 p. 112) for the λ_P-calculus. Also adding these constants with these reduction and typing rules to the λ_P-calculus preserves subject reduction (Lem. B.1.3).

Remark 5.2.1 . The terms to unfold a continuation or to uncons a list will be necessary for the translated terms to be well-typed. We translate the $\lambda_{\mathscr{C}}$-calculus into the λ_P-calculus with types in order to get automatically a transformation on Cartesian closed categories that builds a categorical model for the lambda calculus with constructors (in Sec. 5.3). However, if we are not interested in the semantic aspect of the translation, but only in a simulation of the $\lambda_{\mathscr{C}}$-calculus by the lambda-calculus, we can easily chose the untyped λ_P-calculus (or even the pure lambda calculus, Rem. 5.2.2) as target: just remove every type, and all the terms $\downarrow_s$,$\uparrow_s$,$\downarrow_c$ and $\uparrow_c$ in this section.

Some encoding In order to ease the reading of the CPS-translation, we may use the following notations in the λ_P-calculus:

- $let\ \langle\!|x,y|\!\rangle := P\ in\ M$ means $(\lambda xy.M)\ \pi_1(P)\ \pi_2(P)$
  Notice that x and y are bound variables in this term.
- The notation for tuples is recalled p. 114. We also extend the previous notation to any tuple, writing
  $let\ \langle\!|x_1;\dots;x_k|\!\rangle_k := N\ in\ M$ for $(\lambda x_1 \dots x_k.M)\ \pi_1^k(N) \cdots \pi_k^k(N)$.

Those notations have the expected computational behaviour:

$$\begin{array}{rrcl} & let\ \langle\!|x_1,x_2|\!\rangle := \langle\!|N_1,N_2|\!\rangle\ in\ M & \to_p^* & M[x_i := N_i]_{i=1,2} \\ & let\ \langle\!|x_1;\dots;x_k|\!\rangle_k := \langle\!|N_1;\dots;N_k|\!\rangle_k\ in\ M & \to_p^* & M[x_i := N_i]_{i=1}^k \\ \text{If } x,y \notin \mathrm{fv}(M), & let\ \langle\!|x,y|\!\rangle := N\ in\ M[z := \langle\!|x,y|\!\rangle] & \to_p^* & M[z := N] \end{array}$$

Also the following typing rules are derivable

$$\frac{\Gamma, x : B_1, y : B_2 \vdash_p M : B \quad \Gamma \vdash_p N : B_1 \times B_2}{\Gamma \vdash_p let\ \langle\!|x,y|\!\rangle := N\ in\ M : B}$$

$$\frac{\Gamma, x : B_1, \dots x_k : B_k \vdash_p M : B \quad \Gamma \vdash_p N : B_1 \times \cdots \times B_k}{\Gamma \vdash_p let\ \langle\!|x_1,\dots,x_k|\!\rangle_k := N\ in\ M : B}$$

In the (untyped) lambda calculus with constructors, we will use an encoding for the Daimon before translating it in the λ_P-calculus:

- $exit$ denotes $M_e M_e$, where $M_e = \lambda xy.xx$. Notice that

$$exit\ M \ \to^*\ exit\ .$$

Remark 5.2.2 (Constants vs. encoding) . Instead of the lambda-calculus with pairs, we could have chosen the pure lambda-calculus as the target calculus, and

then used the Church-encoding for pairs [Chu41] given p. 65. Nevertheless, surjective pairing ($(\!|\pi_1(M)\ ,\ \pi_2(M)|\!) \to M$) does not hold under this encoding, and so the simulation theorem (Theo. 5.1) would have been weaker. In particular, the rule LamApp would not have been simulated, but only the weak head reduction.

5.2.2 Continuation Passing Style

The CPS-translation is given in Fig. 5.2. It mainly paraphrases the description of the stack abstract machine. Indeed, a $\lambda_{\mathscr{C}}$-term t is translated by a λ_P-term t^* that takes a *continuation* argument (we write k the variables of type C for more clarity), unfold it (with term $\uparrow_c$) to obtain an evaluation context (*i.e.* a pair of a case-binding M_θ and a stack M_π) that it uses to run the stack-machine $\theta \star t \star \pi$. Finally t^* returns the result $|M_\theta \star t \star M_\pi|$ of this evaluation:

$$t^* \ = \ \lambda k.let\ (\!|x_\theta, x_\pi|\!) :=\uparrow_c k\ in\ \underbrace{|x_\theta \star t \star x_\pi|}_{\lambda_P}\ ,$$

$$\text{with } |\underbrace{M_\theta}_{\lambda_P} \star \underbrace{t}_{\lambda_{\mathscr{C}}} \star \underbrace{M_\pi}_{\lambda_P}| \quad \text{inductively defined in Fig. 5.2.}$$

Remember (p. 81) that $|M_\theta \star t \star M_\pi|$ is a λ_P-term where M_θ is the stack in which all the arguments appearing in evaluation position will be pushed (after being translated). Also its definition when t is an application is quite obvious:

$$|M_\theta \star t_1 t_2 \star M_\pi| \ = \ |M_\theta \star t_1 \star \downarrow_s (\!|t_2^*, M_\pi|\!)| \ .$$

If t is an abstraction $\lambda x.u$, the translation relies on the ambiguity of the variable names, that can represent a variable of $\lambda_{\mathscr{C}}$ or of λ_P. Indeed, we expect the evaluation to pop the first element of the stack M_π and use it to replace every occurrence of x in (the result of the evaluation of) u. That is exactly the behaviour that is obtained:

$$|M_\theta \star \lambda x.u \star M_\pi| \ = \ let\ (\!|x, x_{\pi'}|\!) :=\uparrow_s M_\pi\ in\ |M_\theta \star u \star x_{\pi'}| \ .$$

If the ($\lambda_{\mathscr{C}}$-)variable x later comes in head position, we see it as a λ_P-variable (that is then substituted by the first element of M_π), and we give it the new continuation as argument:

$$|M'_\theta \star x \star M'_\pi| \ = \ x\ (\downarrow_c (\!|M'_\theta, M'_\pi|\!))$$

Example 5.2.3 (Evaluation of the identity) .

$$\begin{array}{rcl} |M_\theta \star (\lambda x.x)t \star M_\pi| & = & |M_\theta \star \lambda x.x \star \downarrow_s (\!|t^*, M_\pi|\!)| \\ & = & let\ (\!|x, x_{\pi'}|\!) :=\uparrow_s (\downarrow_s (\!|t^*, M_\pi|\!))\ in\ |M_\theta \star x \star x_{\pi'}| \\ & = & let\ (\!|x, x_{\pi'}|\!) :=\uparrow_s (\downarrow_s (\!|t^*, M_\pi|\!))\ in\ x\ (\downarrow_c (\!|M_\theta, x'_\pi|\!)) \\ & \to_p^* & t^*\ (\downarrow_c (\!|M_\theta, M_\pi|\!)) \end{array}$$

Remark that we could also have chosen a different space of variable names for the λ_P-calculus (say ξ, ξ_0 *etc.*) and mapped every variable x of the $\lambda_{\mathscr{C}}$-calculus to a variable ξ_x of the λ_P-calculus.

Now, to have an idea about how the translation works with constructors and case-constructs, look at the following example:

Example 5.2.4 (Translation of a case construct) . In this example we assume that there are $n = 2$ constructors.

$$|M_\theta \star \{\!|\mathsf{c}_2 \mapsto t|\!\} \cdot \mathsf{c}_2 \star M_\pi| \;=\; |\langle\!|N_1, N_2|\!\rangle \star \mathsf{c}_2 \star M_\pi|,$$

where $N_1 = \bot_D$, and

$$N_2 = \lambda k'.let\ \langle\!|x_{\theta'}, x_{\pi'}|\!\rangle :=\uparrow_c k'\ in\ |M_\theta \star t \star x_{\pi'}|$$

Also, $|M_\theta \star \{\!|\mathsf{c}_2 \mapsto t|\!\} \cdot \mathsf{c}_2 \star M_\pi| =$

$$\begin{array}{rl} & let\ \langle\!|x_1, x_2|\!\rangle := \langle\!|N_1, N_2|\!\rangle\ in\ |\bot_{D^n} \star x_2 \star M_\pi| \\ = & let\ \langle\!|x_1, x_2|\!\rangle := \langle\!|N_1, N_2|\!\rangle\ in\ x_2(\downarrow_c \langle\!|\bot_{D^n}, M_\pi|\!\rangle) \\ \to_p^* & N_2\ (\downarrow_c \langle\!|\bot_{D^n}, M_\pi|\!\rangle) \\ \to_p^* & let\ \langle\!|x_{\theta'}, x_{\pi'}|\!\rangle :=\uparrow_c (\downarrow_c \langle\!|\bot_{D^n}, M_\pi|\!\rangle)\ in\ |M_\theta \star t \star x_{\pi'}| \\ \to_p^* & |M_\theta \star t \star M_\pi| \end{array}$$

The translation of a constructor is given by the abstract machine:

$$|M_\theta \star \mathsf{c}_i \star M_\pi| \;=\; let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := M_\theta\ in\ |\bot_{D^n} \star x_i \star M_\pi|$$

We use $\bot_{D^n}$ to denote any term of type D^n. It can be for instance M_θ, but we write $\bot_{D^n}$to make clear that it will not be used if we translate well-defined terms (*i.e.* terms with no match failure).

The translation of a case construct $\{\!|\phi|\!\} \cdot t$ is then given by

$$|M_\theta \star \{\!|\phi|\!\} \cdot t \star M_\pi| \;=\; |\langle\!|N_1; \cdots; N_n|\!\rangle_n \star t \star M_\pi|$$

where $\begin{cases} N_i = \lambda k'.let\ \langle\!|x_{\theta'}, x_{\pi'}|\!\rangle :=\uparrow_c k'\ in\ |M_\theta \star u_i \star x_{\pi'}| & \text{if } \mathsf{c}_i \mapsto u_i \in \phi \\ N_i = M_{\text{`}} & \text{if } \mathsf{c}_i \notin \mathrm{dom}(\phi) \end{cases}$

Just like for $\bot_{D^n}$, $M_{\text{`}}$ denotes a term of type D that is never used if we translate well-defined terms. However we force its definition as we shall need it in the following (in Sec. 5.3) for the semantic interpretation of undefined terms:

$$M_{\text{`}} \;=\; \lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ \bot_D\ (\downarrow_c \langle\!|\langle \bot_D, \ldots, \bot_D\rangle, x_\pi|\!\rangle)$$

where $\bot_D = let\ \langle\!|x, x'_\pi|\!\rangle := x_\pi\ in\ x.$

Notice that the application-context M_π is used to evaluate t, while M_θ becomes the case-context of the terms u_i's in the branches of ϕ. The idea is that t can interact with π, but not with the case-context θ until ϕ has been "opened" and one of the u_i's has come in head position. This happens when a constructor c_i has been evaluated, and replaced by u_i *coming with its own case-context* θ. That is why the argument for case-context $x_{\theta'}$ is not used in N_i.

Finally the Daimon is translated using the pure lambda term *exit* that has the same computational behaviour.

$$t^* \;=\; \lambda k.let\ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k\ in\ |x_\theta \star t \star x_\pi|\ .$$

$$\begin{array}{rcl}
|M_\theta \star x \star M_\pi| & = & x\ (\downarrow_c \langle\!\langle M_\theta, M_\pi \rangle\!\rangle) \\
|M_\theta \star tu \star M_\pi| & = & |M_\theta \star t \star \downarrow_s \langle\!\langle u^*, M_\pi \rangle\!\rangle| \\
|M_\theta \star \lambda x.t \star M_\pi| & = & let\ \langle\!\langle x, x_{\pi'} \rangle\!\rangle :=\uparrow_s M_\pi\ in\ |M_\theta \star t \star x_{\pi'}| \\
 & & \qquad\qquad (\text{if } x \notin \mathrm{fv}(M_\theta, M_\pi)) \\
|M_\theta \star \mathsf{c}_i \star M_\pi| & = & let\ \langle\!\langle x_1; \ldots; x_n \rangle\!\rangle_n := M_\theta\ in\ |\bot_{D^n} \ \star\ x_i \star M_\pi| \\
|M_\theta \star \{\!|\phi|\!\} \cdot t \star M_\pi| & = & |\langle\!\langle N_1; \cdots; N_n \rangle\!\rangle_n \star t \star M_\pi| \\
\big(\text{where} \quad N_i \;=\; \lambda k'.let\ \langle\!\langle z_\theta, z_\pi \rangle\!\rangle :=\uparrow_c k'\ in\ |M_\theta \star u_i \star z_\pi| & & \text{if } \mathsf{c}_i \mapsto u_i \in \phi \\
\qquad N_i \;=\; M_\cdot & & \text{if } \mathsf{c}_i \notin \mathrm{dom}(\phi)\big) \\
|M_\theta \star \maltese \star M_\pi| & = & exit^*
\end{array}$$

Figure 5.2: Translation of $\lambda_{\mathscr{C}}$-calculus into λ_P-calculus

Well typing of the translation We check that the translation is well defined in the typed λ_P-calculus introduced in Sec. 5.2.1, in the sense that the translation of every $\lambda_{\mathscr{C}}$-term is well typed.

Proposition 5.2.1. *Let t be a $\lambda_{\mathscr{C}}$-term and Γ a context of the λ_P-calculus such that $x : D \in \Gamma$ for each $x \in \mathrm{fv}(t)$ (Remember that $D = C \to R$). Then,*

1. *For any λ_P-terms M_θ and M_π, if $\Gamma \vdash_p M_\theta : D^n$ and $\Gamma \vdash_p M_\pi : S$, then $\Gamma \vdash_p |M_\theta \star t \star M_\pi| : R$*
2. *$\Gamma \vdash_p t^* : D$*

Proof : *The second assertion is a direct consequence of the first one, that is proved by structural induction on t.* □

5.2.3 Correct Simulation

Now we show that the lambda calculus with pairs can actually *simulate* the lambda calculus with constructors *via* the CPS translation:

Theorem 5.1 (λ_P simutates $\lambda_{\mathscr{C}}$)**.** *For any $\lambda_{\mathscr{C}}$-terms t, t',*

$$t \to t' \quad \textit{implies} \quad t^* \to_p^* t'^*\ .$$

We first focus on the case when t is the redex involved in the reduction, we care about a reduction in a strict sub-term of t subsequently.

Lemma 5.2.2. *For any $\lambda_{\mathscr{C}}$-terms t, u and any λ_P-terms M_θ, M_π,*

$$|M_\theta \star t \star M_\pi|\ [x := u^*] \quad \to_p^* \quad |\ M_\theta[x := u^*] \star t[x := u] \star M_\pi[x := u^*]\ |$$

Proof : *By induction on t.* □

Lemma 5.2.3. *Let t, u be two $\lambda_{\mathscr{C}}$-terms, and ϕ, ψ two case-bindings. Then for any λ_P-terms M_θ, M_π,*

$$\begin{array}{rcll}
|M_\theta \star \big((\lambda x.t)u\big) \star M_\pi| & \to_p^+ & |M_\theta \star t[x := u] \star M_\pi| & \\
|M_\theta \star \Cross\!\!\!\!\Cross\ u \star M_\pi| & = & |M_\theta \star \Cross\!\!\!\!\Cross \star M_\pi| & \\
|M_\theta \star \{\!|\phi|\!\} \cdot \lambda x.t \star M_\pi| & = & |M_\theta \star \lambda x.\{\!|\phi|\!\} \cdot t \star M_\pi| & \text{if } x \in \mathrm{fv}(\phi) \\
|M_\theta \star \{\!|\phi|\!\} \cdot tu \star M_\pi| & = & |M_\theta \star (\{\!|\phi|\!\} \cdot t)u \star M_\pi| & \\
|M_\theta \star \{\!|\phi|\!\} \cdot \mathsf{c} \star M_\pi| & \to_p^+ & |M_\theta \star u \star M_\pi| & \text{if } c \mapsto u \in \phi \\
|M_\theta \star \{\!|\phi|\!\} \cdot \Cross\!\!\!\!\Cross \star M_\pi| & = & |M_\theta \star \Cross\!\!\!\!\Cross \star M_\pi| & \\
|M_\theta \star \{\!|\phi|\!\} \cdot \{\!|\psi|\!\} \cdot t \star M_\pi| & = & |M_\theta \star \{\!|\phi \circ \psi|\!\} \cdot t \star M_\pi| &
\end{array}$$

PROOF : *1. Up to α-conversion, we assume $x \notin \mathrm{fv}(M_\theta, M_\pi)$.*

$$\begin{array}{rcl}
|M_\theta \star \big((\lambda x.t)u\big) \star M_\pi| & = & |M_\theta \star (\lambda x.t) \star \langle\!| u^*, M_\pi |\!\rangle| \\
& = & \mathit{let}\ \langle\!| x, x_\pi |\!\rangle := \langle\!| u^*, M_\pi |\!\rangle\ \mathit{in}\ |M_\theta \star t \star x_\pi| \\
& \to_p^+ & |M_\theta \star t \star M_\pi|[x := u^*]
\end{array}$$

Moreover $|M_\theta \star t \star M_\pi|[x := u^*] \to_p^* |M_\theta \star t[x := u] \star M_\pi|$ by Lem. 5.2.2.

2.
$$\begin{array}{rcl}
|M_\theta \star \Cross\!\!\!\!\Cross\ u \star M_\pi| & = & |M_\theta \star \Cross\!\!\!\!\Cross \star \downarrow_s \langle\!| u^*, M_\pi |\!\rangle| \\
& = & \mathit{exit}^* \\
& = & |M_\theta \star \Cross\!\!\!\!\Cross \star M_\pi|
\end{array}$$

3.
$$\begin{array}{rcl}
|M_\theta \star \{\!|\phi|\!\} \cdot \lambda x.t \star M_\pi| & = & |\langle\!| N_1; \cdots; N_n |\!\rangle_n \star \lambda x.t \star M_\pi| \\
& & \qquad\qquad \textit{(with the } N_i\textit{'s expected)} \\
& = & \mathit{let}\ \langle\!| x, x'_\pi |\!\rangle := M_\pi\ \mathit{in}\ |\langle\!| N_1; \cdots; N_n |\!\rangle_n \star t \star x'_\pi| \\
& = & \mathit{let}\ \langle\!| x, x'_\pi |\!\rangle := M_\pi\ \mathit{in}\ |M_\theta \star \{\!|\phi|\!\} \cdot t \star x'_\pi| \\
& = & |M_\theta \star \lambda x.\{\!|\phi|\!\} \cdot t \star M_\pi|
\end{array}$$

The second equation holds since $x \notin \mathrm{fv}(\phi)$ implies $x \notin \mathrm{fv}(\langle\!| N_1; \cdots; N_n |\!\rangle_n)$.

4. $|M_\theta \star \{\!|\phi|\!\} \cdot tu \star M_\pi| = |M_\theta \star (\{\!|\phi|\!\} \cdot t)\ u \star M_\pi|$ by definition (same as previous case).

5. Let $\phi = \{\mathsf{c}_j \mapsto u_j / j \in J\}$.
Then $|M_\theta \star \{\!|\phi|\!\} \cdot \mathsf{c}_i \star M_\pi| =$

$$\begin{array}{rcl}
& & |\langle\!| N_1; \cdots; N_n |\!\rangle_n \star \mathsf{c}_i \star M_\pi| \qquad \textit{(with the } N_j\textit{s expected)} \\
& = & \mathit{let}\ \langle\!| x_1; \ldots; x_n |\!\rangle_n := \langle\!| N_1; \cdots; N_n |\!\rangle_n\ \mathit{in}\ |\bot_{D^n} \star\ x_i \star M_\pi| \\
& = & \mathit{let}\ \langle\!| x_1; \ldots; x_n |\!\rangle_n := \langle\!| N_1; \cdots; N_n |\!\rangle_n\ \mathit{in}\ x_i\ \langle\!| \bot_{D^n}\ , M_\pi |\!\rangle \\
& \to_p^+ & N_i\ \langle\!| \bot_{D^n}\ , M_\pi |\!\rangle
\end{array}$$

where $N_i = \lambda y.\mathit{let}\ \langle\!| x_\theta, x_\pi |\!\rangle := y\ \mathit{in}\ |M_\theta \star u_i \star x_\pi|$. Also

$$\begin{array}{rcl}
N_i\ \langle\!| \bot_{D^n}\ , M_\pi |\!\rangle & \to_p & \mathit{let}\ \langle\!| x_\theta, x_\pi |\!\rangle := \langle\!| \bot_{D^n}\ , M_\pi |\!\rangle\ \mathit{in}\ |M_\theta \star u_i \star x_\pi| \\
& \to_p^* & |M_\theta \star u_i \star M_\pi|
\end{array}$$

6.
$$\begin{array}{rcl}
|M_\theta \star \{\!|\phi|\!\} \cdot \Cross\!\!\!\!\Cross \star M_\pi| & = & |\langle\!| N_1; \cdots; N_n |\!\rangle_n \star \Cross\!\!\!\!\Cross \star M_\pi| \\
& & \qquad\qquad \textit{(still with the } N_j\textit{'s expected)} \\
& = & \mathit{exit}^* \\
& = & |M_\theta \star \Cross\!\!\!\!\Cross \star M_\pi|
\end{array}$$

7. Let $\phi = \{\mathsf{c}_i \mapsto t_i / i \in I\}$ and $\psi = \{\mathsf{c}_j \mapsto u_j / j \in J\}$.

$$|M_\theta \star \{\!|\phi|\!\} \cdot \{\!|\psi|\!\} \cdot t \star M_\pi| = |\langle\!| M_1; \cdots; M_n |\!\rangle_n \star \{\!|\psi|\!\} \cdot t \star M_\pi|$$

$$\text{where } M_i = \begin{cases} \lambda y.\mathit{let}\ \langle\!| x_\theta, x_\pi |\!\rangle := y\ \mathit{in}\ |M_\theta \star t_i \star x_\pi| & \text{if } i \in I \\ M_\cdot & \text{if } i \notin I \end{cases}$$

$$|M_\theta \star \{\!|\phi|\!\} \cdot \{\!|\psi|\!\} \cdot t \star M_\pi| = |\langle\!| N_1; \cdots; N_n |\!\rangle_n \star t \star M_\pi|$$

$$\text{where } N_j = \begin{cases} \lambda y'.let\ \langle\!\langle x'_\theta, x'_\pi \rangle\!\rangle := y'\ in\ |\langle\!\langle M_1; \cdots ; M_n \rangle\!\rangle_n \star u_j \star x'_\pi| & \text{if } j \in J \\ M_\diamond & \text{if } j \notin J \end{cases}$$

On the other hand, $|M_\theta \star \{\!\{\phi \circ \psi\}\!\} \cdot t \star M_\pi| \quad = \quad |\langle\!\langle N'_1; \cdots ; N'_n \rangle\!\rangle_n \star t \star M_\pi|$

$$\text{where } N'_j = \begin{cases} \lambda y'.let\ \langle\!\langle x'_\theta, x'_\pi \rangle\!\rangle := y'\ in\ |M_\theta \star \{\!\{\phi\}\!\} \cdot u_j \star x'_\pi| & \text{if } j \in J \\ M_\diamond & \text{if } j \notin J \end{cases}$$

If $j \in J$, $N'_j = \lambda y'.let\ \langle\!\langle x'_\theta, x'_\pi \rangle\!\rangle := y'\ in\ |\langle\!\langle M'_1; \cdots ; M'_n \rangle\!\rangle_n \star u_j \star x'_\pi|$

$$\text{where } M'_i = \begin{cases} \lambda y.let\ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle := y\ in\ |M_\theta \star t_i \star x_\pi| & \text{if } i \in I \\ M_\diamond & \text{if } i \notin I \end{cases}$$

Hence $|M_\theta \star \{\!\{\phi \circ \psi\}\!\} \cdot t \star M_\pi| = |M_\theta \star \{\!\{\phi\}\!\} \cdot \{\!\{\psi\}\!\} \cdot t \star M_\pi|$. □

Converse simulation with strategies CPS-translations were introduced to transform a *call-by-name* (*c.b.n.* for short) calculus into a *call-by-value* (*c.b.v.*) calculus. If we restrict the lambda calculus with constructors to the *weak* strategy $\rightarrowtriangle_{\mathrm{W}}$ (defined in Sec. 5.1.3), and the λ_P-calculus to the *c.b.v.* strategy $\rightarrowtriangle_v$ (recalled in appendix B.1), then the CPS-translation we have defined is a *two-sided* simulation:

1. If t reduces weakly on t', then for any λ_P-terms M_θ, M_π,
$$|M_\theta \star t \star M_\pi| \rightarrowtriangle_v^* |M_\theta \star t' \star M_\pi|$$
2. If $|x_\theta \star t \star x_\pi| \rightarrowtriangle_v N$ then t reduces weakly on a term t' and
$$N \rightarrowtriangle_v^* |x_\theta \star t' \star x_\pi| .$$

We do not detail the proof, but it is based on the one of Lem. 5.2.3 with the remark that for all M, N_1, N_2,

$$let\ \langle\!\langle x_1, x_2 \rangle\!\rangle := \langle\!\langle N_1, N_2 \rangle\!\rangle\ in\ M \quad \rightarrowtriangle_v^* \quad M[x_i := N_i] .$$

Simulation result We now extend the result of Lem. 5.2.3 to reductions that occur in a strict sub-term.

Lemma 5.2.4. *For any $\lambda_{\mathscr{C}}$-term t (with $x \notin \mathrm{fv}(t)$), and any λ_P-terms M_θ, M_π and N,*

$$|M_\theta \star t \star M_\pi|\ [x := N] \quad = \quad |\ M_\theta[x := N] \star t \star M_\pi[x := N]\ | \tag{5.1}$$

For any $\lambda_{\mathscr{C}}$-term t and any λ_P-terms M_θ, M_π,

$$M_\theta \rightarrow_p M'_\theta \quad \Longrightarrow \quad |M_\theta \star t \star M_\pi| \rightarrow_p |M'_\theta \star t \star M_\pi| \tag{5.2}$$
$$M_\pi \rightarrow_p M'_\pi \quad \Longrightarrow \quad |M_\theta \star t \star M_\pi| \rightarrow_p^* |M_\theta \star t \star M'_\pi| \tag{5.3}$$

Proof : *By induction on t.* □

Corollary 5.2.5. *For any $\lambda_{\mathscr{C}}$-terms t, t' and any λ_P-terms M_θ, M_π,*

$$t \rightarrow t' \quad implies \quad |M_\theta \star t \star M_\pi| \rightarrow_p^* |M_\theta \star t' \star M_\pi| .$$

PROOF : *By structural induction on t. In this proof we use the convention that s' denotes a reduct (in one step) of s, for any $\lambda_{\mathscr{C}}$-term s.*

$t = \lambda x.u$:

* *If $u = t'x$, with $x \notin \mathrm{fv}(t')$ (i.e. $t \to_{\mathrm{AL}} t'$) then*

$$|M_\theta \star t \star M_\pi| \quad = \quad let\ \langle\!\langle x, x_\pi \rangle\!\rangle := M_\pi\ in\ |M_\theta \star t' \star \langle\!\langle x^*, x_\pi \rangle\!\rangle| \ .$$

Notice that
$$\begin{array}{rcl} x^* & = & \lambda y.\ let\ \langle\!\langle y_\theta, y_\pi \rangle\!\rangle := y\ in\ x\langle\!\langle y_\theta, y_\pi \rangle\!\rangle \\ & \to_p^* & \lambda y.x\langle\!\langle \pi_1(y), \pi_2(y) \rangle\!\rangle \\ & \to_p & \lambda y.xy \\ & \to_p & x \end{array}$$

Hence,
$$\begin{array}{rcll} |M_\theta \star t \star M_\pi| & \to_p^+ & let\ \langle\!\langle x, x_\pi \rangle\!\rangle := M_\pi\ in\ |M_\theta \star t' \star \langle\!\langle x, x_\pi \rangle\!\rangle| & (5.3) \\ & \to_p^* & |M_\theta \star t' \star \langle\!\langle \pi_1(M_\pi), \pi_2(M_\pi) \rangle\!\rangle| & (5.1) \\ & \to_p^* & |M_\theta \star t' \star M_\pi| & (5.3) \end{array}$$

* *If $u = \maltese$ and $t' = \maltese$ (i.e. $t \to_{\mathrm{LD}} t'$), then*

$$\begin{array}{rcl} |M_\theta \star t \star M_\pi| & = & let\ \langle\!\langle x, x_\pi \rangle\!\rangle := M_\pi\ in\ exit^* \\ & \to_p^2 & exit^* \\ & = & |M_\theta \star t' \star M_\pi| \end{array} \quad .$$

* *If $t' = \lambda x.u'$, then*

$$\begin{array}{rcll} |M_\theta \star t \star M_\pi| & = & let\ \langle\!\langle x, x_\pi \rangle\!\rangle := M_\pi\ in\ |M_\theta \star u \star M_\pi| & \\ & \to_p^* & let\ \langle\!\langle x, x_\pi \rangle\!\rangle := M_\pi\ in\ |M_\theta \star u' \star M_\pi| & (ind.\ hypothesis) \\ & = & |M_\theta \star t' \star M_\pi| & \end{array} \quad .$$

$t = t_1 t_2$:

* *If $t_1 = \lambda x.t_0$ and $t' = t_0[x := t_2]$, or $t_1 = \maltese$ and $t' = \maltese$ we conclude with Lem. 5.2.3.*

* *If $t' = t'_1 t_2$, we conclude with induction hypothesis.*

* *If $t' = t_1 t'_2$, then $|M_\theta \star t \star M_\pi| = |M_\theta \star t_1 \star \langle\!\langle t_2^*, M_\pi \rangle\!\rangle|$.*
But $t_2^* \to_p^* \lambda y.let\ \langle\!\langle y_\theta, y_\pi \rangle\!\rangle := y\ in\ |y_\theta \star t'_2 \star y_\pi|$ by induction hypothesis. Also

$$\begin{array}{rcll} |M_\theta \star t \star M_\pi| & = & |M_\theta \star t_1 \star \langle\!\langle t_2^*, M_\pi \rangle\!\rangle| & \\ & \to_p^+ & |M_\theta \star t_1 \star \langle\!\langle t_2'^*, M_\pi \rangle\!\rangle| & (5.3) \\ & = & |M_\theta \star t' \star M_\pi| & \end{array} \quad .$$

$t = \{\!|\theta|\!\} \cdot u$:

* *If one of the rules* CASECONS, CASEAPP, CASELAM *or* CASECASE *is performed in head position, we conclude with Lem. 5.2.3.*

* *If $t' = \{\!|\theta|\!\} \cdot u'$, we conclude with induction hypothesis.*

* *If $t' = \{\!|\theta'|\!\} \cdot u$, where θ' is $\theta = \{\mathsf{c}_j \mapsto s_j / j \in I\}$ in which one branch $\mathsf{c}_i \mapsto s_i$ has been replaced by $\mathsf{c}_i \mapsto s'_i$, then $|M_\theta \star t \star M_\pi| = |\langle\!|N_1; \cdots; N_n|\!\rangle_n \star u \star M_\pi|$*

$$\text{where } N_j = \begin{cases} \lambda y.let\ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle := y\ in\ |M_\theta \star s_j \star x_\pi| & \text{if } \ j \in I \\ M & \text{if } \ j \notin I \end{cases}$$

By induction hypothesis, $N_i \to_p^* N'_i = \lambda y.let\ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle := y\ in\ |M_\theta \star s'_j \star x_\pi|$.
Hence
$$\begin{array}{rcll} |M_\theta \star t \star M_\pi| & \to_p^* & |\langle\!|N_1; \cdots N'_i \cdots; N_n|\!\rangle_n \star u \star M_\pi| & (5.2) \\ & = & |M_\theta \star \{\!|\theta'|\!\} \cdot u \star M_\pi| & \end{array}$$

□

This achieves the proof of Theo. 5.1 since $t \to t'$ implies

$$\begin{array}{rcl} t^* & = & \lambda k.let\ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k\ in\ |x_\theta \star t \star x_\pi| \\ & \to_p^* & \lambda k.let\ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k\ in\ |x_\theta \star t' \star x_\pi| \\ & = & t'^* \end{array}$$

Consequences on models The theorem of simulation implies (since the $\lambda_{\mathscr{C}}$-calculus satisfies the Church-Rösser property) that if $t \simeq_{\lambda_{\mathscr{C}}} t'$ then $t^* \simeq_p t'^*$. Also every model of the λ_P-calculus constitutes a sound model of the $\lambda_{\mathscr{C}}$-calculus *via* the CPS-translation:

$$\begin{array}{ccccc} \lambda_{\mathscr{C}}\text{-calculus} & \longrightarrow & \lambda_P\text{-calculus} & \longrightarrow & \lambda_P\text{-model} \\ t \simeq_{\lambda_{\mathscr{C}}} t' & & t^* \simeq_p t'^* & & [t^*] = [t'^*] \end{array}$$

The models of the typed lambda-calculus (with pairs) are the Cartesian closed categories [AL91]. Also the CCC that can in addition interpret in a sound way the atomic types and terms introduced in Sec. 5.2.1 provide a model for the lambda calculus with constructors.

In the next section we remark that the *continuation* models for the untyped lambda calculus are such categories, and we explicit how they interpret the $\lambda_{\mathscr{C}}$-calculus. We then check that they are indeed $\lambda_{\mathscr{C}}$-models, in the sense of the definition of the previous chapter.

5.3 Classical model

The untyped lambda calculus can be interpreted in any Cartesian closed category with a *reflexive object* $D \cong D \to D$[1] for the denotations. This equation can be solved in any CCC with two objects R and C such that

$$C \cong C \times R^C \ .$$

Indeed, taking $D = R^C$ we get $D \cong R^{C \times R^C} \cong (R^C)^{R^C}$ by Prop. 4.1.4. This corresponds to the *continuation models* of the lambda calculus[2]. In those semantics, every term (*i.e.* every point of D) is seen as a function taking a *continuation argument* in C and returning a *result* in R. A functional term (*i.e.* a point of D^D) is interpreted in $(R^C)^{R^C} \cong R^{C \times R^C}$, also the continuation argument of a function is a point of $C \times R^C$. It represents a pair composed of *later continuation* (in C) and a term (in R^C) that represents the argument of the function. That is why we can see continuations as *stacks of arguments*, and interpret a term by a process in a stack abstract machine.

[1] This main result of categorical semantic can be found in [AL91, Sec 9.3], [AC03, Sec. 4.6]. It is also informally explained in Sec. 4.2.1.

[2] We might also call them *classical models*, as their underlying logic is the classical logic [LRS93]

5.3.1 Continuation $\lambda_{\mathscr{C}}$-model

To interpret the $\lambda_{\mathscr{C}}$-calculus, we need continuations that are not only composed by a stack of arguments, but also by a case-binding (*cf.* Sec. 5.2). Thus in our model we also need an object S (for stacks of arguments). Case-bindings are represented by n-tuples, *i.e.* points of D^n, also the following equations are required:

$$\begin{cases} C & \cong & D^n \times S & \text{(a continuation consists in a case-binding and a stack)} \\ S & \cong & D \times S & \text{(a stack is made up of terms)} \\ D & = & R^C & \text{(a denotation is a function on continuations that returns a result)} \end{cases}$$

We call a *continuation $\lambda_{\mathscr{C}}$-model* (or *classical $\lambda_{\mathscr{C}}$*-model) a CCC with objects C, R and S satisfying those isomorphisms.

$$C \underset{\texttt{fold}}{\overset{\texttt{unfold}}{\cong}} D^n \times S \qquad ; \qquad S \underset{\texttt{cons}}{\overset{\texttt{uncons}}{\cong}} D \times S$$

Remark that if $\mathbb{C}$ is such a continuation $\lambda_{\mathscr{C}}$-model, then the lambda calculus generated by $\mathbb{C}$ (written $\lambda_{\mathbb{C}}$, *cf.* Sec. B.2) is a good target calculus for the CPS-translation (*cf.* Sec. 5.2.1). Indeed, in the $\lambda_{\mathbb{C}}$-calculus, C, R and S are the atomic types for the objects C, R and S of $\mathbb{C}$, the terms $\uparrow_c$ and $\downarrow_c$ are the atomic terms for `unfold` and `fold`, and the terms $\uparrow_s$ and $\downarrow_s$ are the atomic terms for `uncons` and `cons`. Then by definition of the $\lambda_{\mathbb{C}}$-calculus,

$$\begin{array}{lll} \vdash_p \uparrow_s : & S \to (D \times S) & \lambda x.\, \downarrow_s (\uparrow_s x) \twoheadrightarrow_p \lambda x.x \\ \vdash_p \downarrow_s : & (D \times S) \to S & \lambda x.\, \uparrow_s (\downarrow_s x) \twoheadrightarrow_p \lambda x.x \end{array}$$

$$\begin{array}{lll} \vdash_p \uparrow_c : & C \to (D^n \times S) & \lambda x.\, \downarrow_c (\uparrow_c x) \twoheadrightarrow_p \lambda x.x \\ \vdash_p \downarrow_c : & (D^n \times S) \to C & \lambda x.\, \uparrow_c (\downarrow_c x) \twoheadrightarrow_p \lambda x.x \end{array}$$

The following proposition completes the other reductions that we need in the target calculus:

Proposition 5.3.1. *For any λ_P-term N, the following reduction are admissible (Def. B.2.1):*

$$\begin{array}{llll} \downarrow_s (\uparrow_s N) & \to_{\mathbb{C}} & N & \qquad \uparrow_s (\downarrow_s N) \to_{\mathbb{C}} N \\ \downarrow_c (\uparrow_c N) & \to_{\mathbb{C}} & N & \qquad \uparrow_c (\downarrow_c N) \to_{\mathbb{C}} N \end{array}$$

Proof : *We prove the admissibility of the first reduction, the other ones are proved by the same method. Assume there is a context Γ and a type B such that $\Gamma \vdash_p \downarrow_s (\uparrow_s N) : B$. Then it means that $\Gamma \vdash_p \downarrow_s : A' \to B'$ and $\Gamma \vdash_p \uparrow_s N : A'$ for some A' and some B' such that $B' = B$. Moreover, $\Gamma \vdash_p \downarrow_s : A' \to B'$ implies $A' \to B' = (D \times S) \to S$ (since $\downarrow_s$ can only be typed with the rule $\mathbf{ax_T}$ or $\mathbf{subs}$), which means $A' = D \times S$ and $B' = S$ by good design of the parameter $\mathcal{R}$ in the $\lambda_{\mathbb{C}}$-calculus (Lem. B.2.1). Thus $\Gamma \vdash_p \uparrow_s N : D \times S$, and we conclude in the same way (using $\Gamma \vdash_p \uparrow_s : S \to (D \times S)$) that $\Gamma \vdash_p N : S$. Also we can derive $\Gamma \vdash_p (\lambda x.\, \downarrow_s (\uparrow_s x))N : S$*

On the one hand, $(\lambda x.\downarrow_s(\uparrow_s x))\,N \to_p \downarrow_s(\uparrow_s N)$ *so* $\lfloor(\lambda x.\downarrow_s(\uparrow_s x))\,N\rfloor_\Gamma = f_{\downarrow_s(\uparrow_s \mathrm{N})}$ *by Lem. B.2.3. On the other hand,* $(\lambda x.\downarrow_s(\uparrow_s x))\,N \to_p (\lambda x.x)\,N \to_p N$ *and then* $\lfloor(\lambda x.\downarrow_s(\uparrow_s x))N\rfloor_\Gamma = f_{\mathrm{N}}$. *Finally* $f_{\downarrow_s(\uparrow_s \mathrm{N})} = f_{\mathrm{N}}$. □

From continuation λ-model to continuation $\lambda_{\mathscr{C}}$-model Notice that any continuation model of pure lambda calculus provides a continuation $\lambda_{\mathscr{C}}$-model. Indeed, with an isomorphism

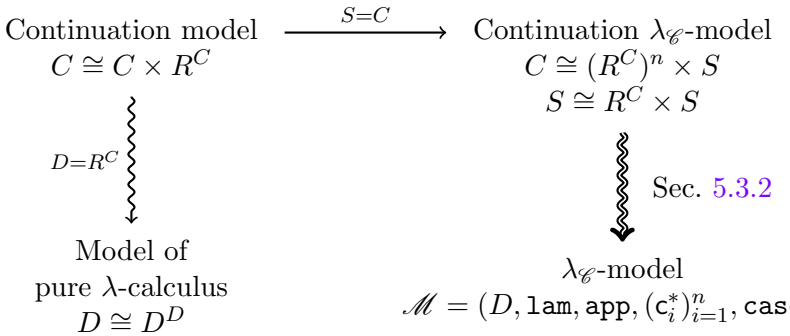
$$C \underset{p}{\overset{i}{\cong}} R^C \times C\ ,$$

we can chose $S = C$, $D = R^C$ and the following morphisms to get a continuation $\lambda_{\mathscr{C}}$-model:

$$C \underset{p}{\overset{i}{\cong}} D \times C \underset{Id\times p}{\overset{Id\times i}{\cong}} D \times (D\times C) \quad \cdots \quad \underset{Id\times(\ldots\times(Id\times p))}{\overset{Id\times(\cdots\times(Id\times i))}{\cong}} D \times (\cdots \times (D\times C)) \underset{\langle(\pi_1;\pi_1^n),\ldots\langle(\pi_1;\pi_n^n),\pi_2\rangle\rangle}{\overset{\langle\langle\pi_1,\ldots,\pi_{2\ldots 21}\rangle,\pi_{2^n}\rangle}{\cong}} D^n \times S$$

$$S \underset{p}{\overset{i}{\cong}} D \times S$$

Also every continuation model induces a continuation $\lambda_{\mathscr{C}}$-model. Next we show that those continuation $\lambda_{\mathscr{C}}$-models are indeed $\lambda_{\mathscr{C}}$-models in sense of Def 4.2.2. This provides a way to construct a $\lambda_{\mathscr{C}}$-model from a classical model of pure lambda calculus.



5.3.2 From continuation $\lambda_{\mathscr{C}}$-models to $\lambda_{\mathscr{C}}$-models

In this Section, $\mathbb{C}$ is a Cartesian closed category with objects R, C and S such that $C \cong D^n \times S$ and $S \cong D \times S$ (where $D = R^C$). We define the morphisms $\mathtt{lam}$, $\mathtt{app}$, $\mathtt{c}_i^*$'s and $\grave{}$ and we prove that they form a $\lambda_{\mathscr{C}}$-model with D. It might

be heavy to express them within the structure of Cartesian closed categories, also we use terms of the lambda-calculus generated by $\mathbb{C}$ (*cf.* appendix B.2), and their image in $\mathbb{C}$ (defined in Fig. B.2) for the definition.

$$
\begin{array}{rcl}
M_{\mathtt{lam}} & = & \lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ \ let\ \langle\!|x, x'_\pi|\!\rangle :=\uparrow_s x_\pi\ in\ z\ x\ \left(\downarrow_c \langle\!|x_\theta, x'_\pi|\!\rangle\right) \\
M_{\mathtt{app}} & = & \lambda x.\lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ z\ \left(\downarrow_c \langle\!|x_\theta, \downarrow_s \langle\!|x, x_\pi|\!\rangle\!|\rangle\right) \\
M_{\mathtt{c}_i} & = & \lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ \ let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := x_\theta\ in\ x_i\ k \\
M_{\mathtt{case}} & = & \lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in \\
 & & let\ \langle\!|y_\phi, y|\!\rangle := z\ in\ y\ \left(\downarrow_c \langle\!|\langle\!|M_1, \ldots, M_n|\!\rangle_n, x_\pi|\!\rangle\right)\ , \\
\text{where } M_i & = & \lambda k'.let\ \langle\!|z_\theta, z_\pi|\!\rangle :=\uparrow_c k'\ in \\
 & & let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := y_\phi\ in\ x_i\ \left(\downarrow_c \langle\!|x_\theta, z_\pi|\!\rangle\right) \\
M_{\backprime} & = & \lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ \bot_D\ \left(\downarrow_c \langle\!|\langle\bot_D, \ldots, \bot_D\rangle, x_\pi|\!\rangle\right) \\
\text{with } \bot_D & = & let\ \langle\!|x, x'_\pi|\!\rangle := x_\pi\ in\ x
\end{array}
$$

Figure 5.3: Terms for the morphisms of a continuation $\lambda_{\mathscr{C}}$-model

The morphisms that are used to build a $\lambda_{\mathscr{C}}$-model in $\mathbb{C}$ correspond to the terms defined in Fig. 5.3. Remark that $M_{\backprime}$ is the same as the one we used in the CPS-translation (p. 85). Moreover, $M_{\mathtt{c}_i} \to_{\mathbb{C}} \mathtt{c}_i^*$:

$$\mathtt{c}_i^* \ = \ \lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := x_\theta\ in\ x_i\ (\downarrow_c \langle\!|\bot_{D^n}, x_\pi|\!\rangle)\ ,$$

where $\bot_{D^n} = x_\theta$. Also both $M_{\mathtt{c}_i}$ and $\mathtt{c}_i^*$ reduce on the same term

$$N = \lambda k.\pi_i^n(\pi_1(\uparrow_c k))\ k\ ,$$

hence $\lfloor \mathtt{c}_i^* \rfloor = f_{\mathrm{N}} = \lfloor M_{\mathtt{c}_i} \rfloor$ by Lem. B.2.3. Thus $M_{\mathtt{c}_i} \to_{\mathbb{C}} \mathtt{c}_i^*$ (and conversely).

Lemma 5.3.2. *The terms defined in Fig. 5.3 are typable with the following derivable judgements:*

$$z : D \to D \vdash_p M_{\mathtt{lam}} : D \tag{5.4}$$

$$z : D \vdash_p M_{\mathtt{app}} : D \to D \tag{5.5}$$

$$\vdash_p M_{\mathtt{c}_i} : D \tag{5.6}$$

$$z : D^n \times D, k : C \vdash_p M_i : D \quad and \quad z : D^n \times D \vdash_p M_{\mathtt{case}} : D \tag{5.7}$$

$$\vdash_p M_{\backprime} : D \tag{5.8}$$

This enables the definition of the morphisms of a $\lambda_{\mathscr{C}}$-model.

Definition 5.3.1

In a continuation $\lambda_{\mathscr{C}}$-model, the morphisms $\mathtt{lam}$, $\mathtt{app}$, c_i^*, $\grave{}$ and $\mathtt{case}$ are given by

$$\begin{array}{rcl} \mathtt{lam} & = & \lfloor M_{\mathtt{lam}} \rfloor_{z:D\to D} : D^D \to D \\ \mathtt{app} & = & \lfloor M_{\mathtt{app}} \rfloor_{z:D} : D \to D^D \\ \mathtt{c}_i^* & = & \lfloor M_{\mathtt{c}_i} \rfloor \ : \mathbf{1} \to D \\ \mathtt{case} & = & \lfloor M_{\mathtt{case}} \rfloor_{z:D^n\times D} : D^n \times D \to D \\ \grave{} & = & \lfloor M_{\grave{}} \rfloor \ : \mathbf{1} \to D \end{array}$$

where $M_{\mathtt{lam}}, M_{\mathtt{app}}, M_{\mathtt{c}_1}, M_{\mathtt{case}}$ and $M_{\grave{}}$ are defined in Fig 5.3.

Now we check that this definition makes the diagrams of Fig. 4.1 p. 59 commute.

Lemma 5.3.3. $\mathtt{app}\ ;\ \mathtt{lam} \ = \ Id_D$.

PROOF : *By* (B.2), $\lceil \mathtt{app};\mathtt{lam} \rceil \to_p^* \lambda z.M_{\mathtt{lam}}[z := M_{\mathtt{app}}]$. *Hence,*

$$\begin{array}{rcl} \lceil \mathtt{app};\mathtt{lam} \rceil & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ \ let\ \langle x, x'_\pi \rangle := x_\pi\ in\ M_{\mathtt{app}}\ x\ \left(\downarrow_c \langle x_\theta, x'_\pi \rangle \right) \\ & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ \ let\ \langle x, x'_\pi \rangle := \uparrow_s x_\pi\ in \\ & & \qquad let\ \langle y_\theta, y_\pi \rangle := \uparrow_c \left(\downarrow_c \langle x_\theta, x'_\pi \rangle \right)\ in\ z\ \left(\downarrow_c \langle y_\theta, \downarrow_s \langle x, y_\pi \rangle \rangle \right) \\ & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ \ let\ \langle x, x'_\pi \rangle := \uparrow_s x_\pi\ in \\ & & \qquad z\ \left(\downarrow_c \langle x_\theta, \downarrow_s \langle x, x'_\pi \rangle \rangle \right) \\ & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ \ let\ \langle x, x'_\pi \rangle := \uparrow_s x_\pi\ in \\ & & \qquad z\ \left(\downarrow_c \langle x_\theta, \downarrow_s \langle x, x'_\pi \rangle \rangle \right) \\ & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ z\ \left(\downarrow_c \langle x_\theta, \downarrow_s (\uparrow_s x_\pi) \rangle \right) \\ & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ z\ \left(\downarrow_c \langle x_\theta, x_\pi \rangle \right) \\ & \to_p^* & \lambda z.\lambda k\ z\ \left(\downarrow_c (\uparrow_c k) \right) \\ & \to_p^* & \lambda z.z \end{array}$$

On the other hand, $\dot{Id}_D \to_p \lambda z.z$ *by definition of* $\mathcal{R}$. *Thus* $\mathtt{app}\ ;\ \mathtt{lam} \ = \ Id_D$ *by* (B.1). □

Lemma 5.3.4. $\mathtt{lam}\ ;\ \mathtt{app} \ = \ Id_{D^D}$.

PROOF : *By* (B.2), $\lceil \mathtt{lam};\mathtt{app} \rceil \to_p^* \lambda z.M_{\mathtt{app}}[z := M_{\mathtt{lam}}]$. *Hence,*

$$\begin{array}{rcl} \lceil \mathtt{lam};\mathtt{app} \rceil & \to_p^* & \lambda z.\lambda x k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ M_{\mathtt{lam}}\left(\downarrow_c \langle x_\theta, \downarrow_s \langle x, x_\pi \rangle \rangle \right) \\ & \to_p^* & \lambda z.\lambda x k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ let\ \langle y_\theta, y_\pi \rangle := \langle x_\theta, \downarrow_s \langle x, x_\pi \rangle \rangle\ in \\ & & \qquad let\ \langle y, y'_\pi \rangle := \uparrow_s y_\pi\ in\ z\ y\ \left(\downarrow_c \langle y_\theta, y'_\pi \rangle \right) \\ & \to_p^* & \lambda z.\lambda x k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in \\ & & \qquad let\ \langle y, y'_\pi \rangle := \langle x, x_\pi \rangle\ in\ z\ y\ \left(\downarrow_c \langle x_\theta, y'_\pi \rangle \right) \\ & \to_p^* & \lambda z.\lambda x k.let\ \langle x_\theta, x_\pi \rangle := \uparrow_c k\ in\ z\ x\ \left(\downarrow_c \langle x_\theta, x_\pi \rangle \right) \\ & \to_p^* & \lambda z x k.z\ x\ k \\ & \to_p^* & \lambda z.z \end{array}$$

On the other hand, $\dot{Id}_{D^D} \to_p \lambda z.z$, *thus* $\mathtt{app}\ ;\ \mathtt{lam} \ = \ Id_D$ *by* (B.1). □

The following lemma ensures the commutation of the diagram $(D2)$.

Lemma 5.3.5. *For all* $i \leq n$, $\pi_2\ ;\ \pi_i^n \ = \ (Id_{D^n} \times \mathtt{c}_i^*)\ ;\ \mathtt{case} : \ D^n \times \mathbf{1} \to D^n \times D$.

PROOF : *On one hand,* $\lceil \pi_2 \mathbin{;} \pi_i^n \rceil = \lambda z.(\lambda x.\pi_i^n(x))\big((\lambda x.\pi_2(x))\ z\big) \to_p^* \lambda z.\pi_i^n(\pi_2(z))$

On the other hand,

$$(Id_{D^n} \times \mathsf{c}_i^*) = \langle \pi_1 \, , (\pi_2; \mathsf{c}_i^*) \rangle = \lfloor \{\!|\pi_1(x), M_{\mathsf{c}_i}|\!\} \rfloor_{x:D^n \times \mathbf{1}} : \ D^n \times \mathbf{1} \to D^n \times D \ .$$

So $\lceil (Id_{D^n} \times \dot{\mathsf{c}}_i^*) \ ; \ \mathtt{case} \rceil \to_p^* \lambda x.M_{\mathtt{case}}[z := \{\!|\pi_1(x), M_{\mathsf{c}_i}|\!\}]$ *by* (B.2).
Hence

$$\begin{array}{rcl}
\lceil (Id_{D^n} \times \dot{\mathsf{c}}_i^*) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x.\lambda k.let\ \{\!|x_\theta, x_\pi|\!\} :=\uparrow_c k\ in\ let\ \{\!|z_\theta, y|\!\} := \{\!|\pi_1(x), M_{\mathsf{c}_i}|\!\}\ in \\
& & \qquad y\ \big(\downarrow_c \{\!|\{\!|M_1, \ldots, M_n|\!\}_n, x_\pi|\!\}\big)\ , \\
\text{where } M_i & = & \lambda k'.let\ \{\!|y_\theta, y_\pi|\!\} :=\uparrow_c k'\ in \\
& & \qquad let\ \{\!|x_1; \ldots; x_n|\!\}_n := z_\theta\ in\ x_i\ \big(\downarrow_c \{\!|x_\theta, y_\pi|\!\}\big) \\
\lceil (Id_{D^n} \times \dot{\mathsf{c}}_i^*) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x.\lambda k.let\ \{\!|x_\theta, x_\pi|\!\} :=\uparrow_c k\ in \\
& & \qquad M_{\mathsf{c}_i}\ \big(\downarrow_c \{\!|\{\!|M'_1, \ldots, M'_n|\!\}_n, x_\pi|\!\}\big)\ , \\
\text{where } M'_i & = & \lambda k'.let\ \{\!|y_\theta, y_\pi|\!\} :=\uparrow_c k'\ in \\
& & \qquad let\ \{\!|x_1; \ldots; x_n|\!\}_n := \pi_1(x)\ in\ x_i\ \big(\downarrow_c \{\!|x_\theta, y_\pi|\!\}\big) \\
\lceil (Id_{D^n} \times \dot{\mathsf{c}}_i^*) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x.\lambda k.let\ \{\!|x_\theta, x_\pi|\!\} :=\uparrow_c k\ in \\
& & \quad let\ \{\!|x_\theta, x_\pi|\!\} := \{\!|\{\!|M'_1, \ldots, M'_n|\!\}_n, x_\pi|\!\}\ in \\
& & \qquad let\ \{\!|x_1; \ldots; x_n|\!\}_n := x_\theta\ in\ x_i\ k \\
& \to_p^* & \lambda x.\lambda k.let\ \{\!|x_\theta, x_\pi|\!\} :=\uparrow_c k\ in \\
& & \qquad let\ \{\!|x_1; \ldots; x_n|\!\}_n := \{\!|M'_1, \ldots, M'_n|\!\}_n\ in\ x_i\ k \\
& \to_p^* & \lambda x.\lambda k.let\ \{\!|x_\theta, x_\pi|\!\} :=\uparrow_c k\ in\ M'_i\ k \\
& \to_p^* & \lambda x.\lambda k.let\ \{\!|x_\theta, x_\pi|\!\} :=\uparrow_c k\ in\ let\ \{\!|y_\theta, y_\pi|\!\} :=\uparrow_c k\ in \\
& & \qquad let\ \{\!|x_1; \ldots; x_n|\!\}_n := \pi_1(x)\ in\ x_i\ \big(\downarrow_c \{\!|x_\theta, y_\pi|\!\}\big) \\
& \to_p^* & \lambda x.\lambda k.let\ \{\!|x_1; \ldots; x_n|\!\}_n := \pi_1(x)\ in \\
& & \qquad x_i\ \big(\downarrow_c \{\!|\pi_1(\uparrow_c k), \pi_2(\uparrow_c k)|\!\}\big) \\
& \to_p^* & \lambda x.\lambda k.let\ \{\!|x_1; \ldots; x_n|\!\}_n := \pi_1(x)\ in\ x_i\ k \\
& \to_p^* & \lambda x.\pi_i^n(\pi_1(x))
\end{array}$$

Hence $\pi_2 \ ; \ \pi_i^n = (Id_{D^n} \times \mathsf{c}_i^*) \ ; \ \mathtt{case}$ *by* (B.1). □

The following lemma ensures the commutation of the diagram $(D3)$.

Lemma 5.3.6. *Let* $M_{\cong} = \{\!|\pi_{11}(x), \{\!|\pi_{21}(x), \pi_2(x)|\!\}|\!\}$.
The typing judgement $x : (D^n \times D) \times D \vdash_p M_{\cong} : D^n \times (D \times D)$ *is derivable. Write* $h_{\cong} = \lfloor M_{\cong} \rfloor_{x:(D^n \times D) \times D}$. *It is commutation isomorphism from* $(D^n \times D) \times D$ *to* $D^n \times (D \times D)$. *Then,*

$$(\mathtt{case} \times Id_D); (\mathtt{app} \times Id_D); \mathtt{ev} \ = \ h_{\cong}; (Id_{D^n} \times (\mathtt{app} \times Id_D)); (Id_{D^n} \times \mathtt{ev}); \mathtt{case} \ .$$

PROOF : *cf. appendix A.2.2.* □

For the commutation of $(D5)$, remember that $\bullet = \langle \ldots, ((Id_{D^n} \times \pi_i^n); \mathtt{case}), \ldots \rangle$.

Lemma 5.3.7. *Write* $M_{\cong} = \{\!|\pi_{11}(z), \{\!|\pi_{21}(z), \pi_2(z)|\!\}|\!\}$,
and $h_{\cong} = \lfloor M_{eq} \rfloor_{z:(D^n \times D^n) \times D}$. *It is the commutation isomorphism from* $(D^n \times D^n) \times D$ *to* $D^n \times (D^n \times D)$. *Then,*

$$(\bullet \times Id_D) \ ; \ \mathtt{case} \ = \ h_{\cong} \ ; \ (Id_{D^n} \times \mathtt{case}) \ ; \ \mathtt{case} \ .$$

PROOF : *cf. appendix A.2.2.* □

Finally, the following lemma ensures that the diagram $(D6)$ commutes:

Lemma 5.3.8. *In $D^n \times \mathbf{1} \to D$ the following equality holds:*

$$(Id_{D^n} \times \grave{}) \ ; \ \mathtt{case} \quad = \quad \pi_2 \ ; \ \grave{} \ .$$

PROOF : *We show that the atomic terms associated to these morphisms reduce on a same term.*

$$\begin{array}{rcll}
\lceil Id_{D^n} \times \grave{} \rceil & \to_p^* & \lambda x.\langle\!\langle \pi_1(x), M_{\grave{}} \rangle\!\rangle & (B.7) \\
\lceil (Id_{D^n} \times \grave{}) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x.M_{\mathtt{case}}[z := \langle\!\langle \pi_1(x), M_{\grave{}} \rangle\!\rangle] & (B.2) \\
& = & \lambda x.\lambda k.let \ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k \ in & \\
& & \quad let \ \langle\!\langle y_\phi, y \rangle\!\rangle := \langle\!\langle \pi_1(x), M_{\grave{}} \rangle\!\rangle \ in \ y \ \left(\downarrow_c \langle\!\langle \langle\!\langle M_1, \dots, M_n \rangle\!\rangle_n, x_\pi \rangle\!\rangle \right) , & \\
\quad where \ M_i & = & \lambda k'.let \ \langle\!\langle z_\theta, z_\pi \rangle\!\rangle :=\uparrow_c k' \ in & \\
& & \quad let \ \langle\!\langle x_1; \dots; x_n \rangle\!\rangle_n := y_\phi \ in \ x_i \ \left(\downarrow_c \langle\!\langle x_\theta, z_\pi \rangle\!\rangle \right) & \\
\\
\lceil (Id_{D^n} \times \dot{\grave{}}) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x k.let \ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k \ in & \\
& & \quad M_{\grave{}} \ \left(\downarrow_c \langle\!\langle \langle\!\langle M'_1, \dots, M'_n \rangle\!\rangle_n, x_\pi \rangle\!\rangle \right) , & \\
\quad where \ M'_i & = & \lambda k'.let \ \langle\!\langle z_\theta, z_\pi \rangle\!\rangle :=\uparrow_c k' \ in & \\
& & \quad let \ \langle\!\langle x_1; \dots; x_n \rangle\!\rangle_n := \pi_1(x) \ in \ x_i \ \left(\downarrow_c \langle\!\langle x_\theta, z_\pi \rangle\!\rangle \right) & \\
\\
\lceil (Id_{D^n} \times \dot{\grave{}}) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x k.let \ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k \ in & \\
& & \quad let \ \langle\!\langle y_\theta, y_\pi \rangle\!\rangle := \langle\!\langle \langle\!\langle M'_1, \dots, M'_n \rangle\!\rangle_n, x_\pi \rangle\!\rangle \ in & \\
& & \quad\quad \bot'_D \ \left(\downarrow_c \langle\!\langle \langle \bot'_D, \dots, \bot'_D \rangle, y_\pi \rangle\!\rangle \right) & \\
\quad where \ \bot'_D & = & let \ \langle\!\langle y, y'_\pi \rangle\!\rangle := y_\pi \ in \ y & \\
\\
\lceil (Id_{D^n} \times \dot{\grave{}}) \ ; \ \mathtt{case} \rceil & \to_p^* & \lambda x k.let \ \langle\!\langle x_\theta, x_\pi \rangle\!\rangle :=\uparrow_c k \ in & \\
& & \quad \bot_D \ \left(\downarrow_c \langle\!\langle \langle \bot_D, \dots, \bot_D \rangle, x_\pi \rangle\!\rangle \right) & \\
\quad where \ \bot_D & = & let \ \langle\!\langle y, y'_\pi \rangle\!\rangle := x_\pi \ in \ y &
\end{array}$$

This means $\lceil (Id_{D^n} \times \dot{\grave{}}) \ ; \ \mathtt{case} \rceil \to_p^* \lambda x.M_{\grave{}}$.
By (B.2), $\lceil \pi_2 \ ; \ \grave{} \rceil \to_p^* \lambda x.M_{\grave{}}$ (since $M_{\grave{}}$ is a closed term).
Also $(Id_{D^n} \times \grave{}) \ ; \ \mathtt{case} = \pi_2 \ ; \ \grave{}$. □

With all these lemmas we can conclude that any continuation model can be turned into a $\lambda_{\mathscr{C}}$-model.

Theorem 5.2. *Given $\mathbb{C}$ a CCC with three objects R, C and S such that*

$$C \cong (R^C)^n \times S \qquad \text{and} \qquad S \cong R^C \times S \ ,$$

let $\mathscr{M}_{cl} = (\mathbb{C}, D, \mathtt{app}, \mathtt{lam}, (\mathtt{c}_i^*)_{i=1}^n, \grave{})$ *where*

- $D = R^C$,
- $\mathtt{app}$, $\mathtt{lam}$, $(\mathtt{c}_i^*)_{i=1}^n$ *and* $\grave{}$ *are defined as in Def. 5.3.1.*

Then $\mathscr{M}_{cl}$ is a $\lambda_{\mathscr{C}}$-model in $\mathbb{C}$, in the sense of Def. 4.2.2.

PROOF : *By Lem. 5.3.2, $\mathtt{app}, \mathtt{lam}, \grave{}$ and the $\mathtt{c}_i^*$'s are all well defined morphisms between the required objects. Moreover, the diagram $(D1)$ commutes in $\mathbb{C}$ by Lem. 5.3.3 and*

Lem. 5.3.4. The diagram $(D3)$ *commutes by Lem. 5.3.6, which entails the commutation of* $(D4)$ *by Prop. 4.2.1. Finally, the commutations of the diagrams* $(D2)$, $(D3)$ *and* $(D6)$ *are ensured by Lem. 5.3.5, Lem. 5.3.7, and Lem. 5.3.8 respectively.* □

We call $\mathscr{M}_{cl}$ the *classical* $\lambda_{\mathscr{C}}$*-model* in the category $\mathbb{C}$.

5.3.3 A non syntactic model of the $\lambda_{\mathscr{C}}$-calculus

Searching for a denotational semantics for the lambda calculus, Dana Scott elaborated in the late 60's a theory of *domains*, in which he developed a method [Sco70, GS90] to embed any complete lattice D in a lattice D_∞ satisfying

$$D_\infty \;\cong\; D_\infty \to D_\infty \;.$$

Domain theory emerged as a fruitful setting for the denotational semantics of various families of lambda calculi, and has been well studied since then [SHLG94, AC03].

Scott's construction also works for *complete partial orders* (*cpo*, for short), and enables to solve various recursive equations [Win93, Chap. 12] written with the unit object $\mathbf{1}$, the product, the functional arrow and the sum.

In particular for a given *cpo* R, the equation $X \cong X \times R^X$ has a least solution C. It is thus possible to construct a continuation model in $\mathbb{C}$po, the category of complete partial orders and continuous functions. As explained p. 92 this provides a continuation $\lambda_{\mathscr{C}}$-model, by taking $S = C$. By Theo. 5.2 we know then how to define the classical model $\mathscr{M}_{cl}$ of the lambda calculus with constructors in $\mathbb{C}$po. Unlike the PER model $\mathscr{M}_{synt}$ (Sec 4.3), it is not a syntactic model.

Streicher and Reus have shown [RS98] that every Scott's D_∞-model is isomorphic to a continuation model R^C (where R is in fact the initial *cpo* used for the construction of D_∞). Also there is few hope to construct a $\lambda_{\mathscr{C}}$-model in a category of partial orders that would not be equivalent to the classical model.

However continuation $\lambda_{\mathscr{C}}$-models are probably not complete for the lambda calculus with constructors: there might be some $\lambda_{\mathscr{C}}$-models that are not equivalent to the classical model. Hofmann and Streicher have shown [HS97] that continuation models are complete for the $\lambda\mu$-calculus of Parigot [Par92]. Also we conjecture that the continuation $\lambda_{\mathscr{C}}$-models are complete for the $\lambda\mu$-calculus with constructors (obtained by merging both calculi).

Appendix A

Some detailed proofs

A.1 Categorical models

A.1.1 Proof of soundness of $\lambda_{\mathscr{C}}$-models

Lemma (4.2.4). *Given* $\Gamma = \{x_1, \dots, x_k\}$, *and two terms* t *and* u *such that* $\mathrm{fv}(u) \subseteq \Gamma$ *and* $\mathrm{fv}(t) \subseteq \Gamma \cup \{y\}$,

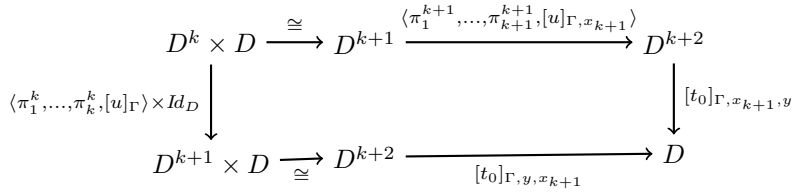
$$[t[y := u]]_\Gamma \;=\; D^k \xrightarrow{\langle Id, [u]_\Gamma\rangle} D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t]_{\Gamma,y}} D$$

Proof : *Remember that the equivalence morphism from* $D^k \times D$ *to* D^{k+1} *is* $\langle(\pi_1; \pi_1^k), \dots, (\pi_1 \pi_k^k), \pi_2\rangle$. *Also what we want to show is*

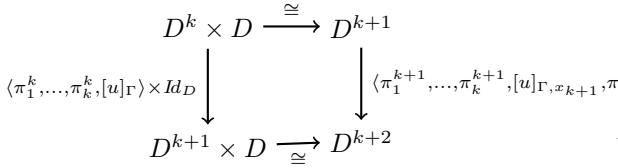
$$[t[y := u]]_\Gamma \;=\; D^k \xrightarrow{\langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma\rangle} D^{k+1} \xrightarrow{[t]_{\Gamma,y}} D \ .$$

We do it by structural induction on t*:*

- *If* $t = y$, $[t]_{\Gamma,y} = \pi_{k+1}^{k+1}$ *so* $[u]_\Gamma = \langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma\rangle; [t]_{\Gamma,y}$.
- *If* $t = x_i$, $[t]_{\Gamma,y} = \pi_i^{k+1}$ *so* $[x_i]_\Gamma = \pi_i^k = \langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma\rangle; [t]_{\Gamma,y}$.
- *If* $t = \mathtt{c}$, $[t]_{\Gamma,y} = !_{D^{k+1}}; c^*$ *and* $\langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma\rangle; !_{D^{k+1}} = !_{D^k}$ *by uniqueness of the morphism in* $\mathbf{1}$. *Thus the equality holds.*
- *If* $t = \lambda x_{k+1}.t_0$, *then* $[\lambda x.t_0[y := u]]_\Gamma = \Lambda(f); \mathtt{lam}$,
  with $f = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t_0[y:=u]]_{\Gamma, x_{k+1}}} D$. *By induction hypothesis,*
  $f = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{\langle \pi_1^{k+1}, \dots, \pi_{k+1}^{k+1}, [u]_{\Gamma, x_{k+1}}\rangle} D^{k+2} \xrightarrow{[t_0]_{\Gamma, x_{k+1}, y}} D$.
  Since $[\lambda x_{k+1}.t_0]_{\Gamma,y} = \Lambda(f'); \mathtt{lam}$, *with* $f' = \; D^{k+1} \times D \xrightarrow{\cong} D^{k+2} \xrightarrow{[t_0]_{\Gamma, y, x_{k+1}}} D$,
  we just have to prove that $\Lambda(f) = \langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma\rangle; \Lambda(f')$.
  By Lem. 4.1.3, this equation is equivalent to $f = (\langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma\rangle \times Id_D); f'$. *We prove that the following diagram commutes:*

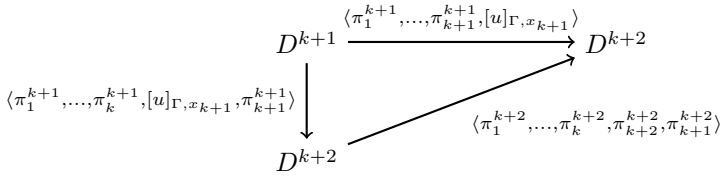


It does commute, as merger of the three following diagrams:

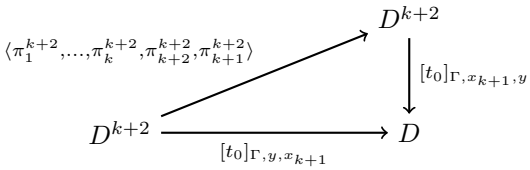


This diagram commutes by weakening property (Lem. 4.2.3) and by uniqueness of product morphism.

This diagram commutes by uniqueness of product morphism.



This diagram commutes by exchange property (Lem. 4.2.3).



- *If* $t = t_1 t_2$, *then* $[t[y := u]]_\Gamma = \langle [t_1[y := u]]_\Gamma, [t_2[y := u]]_\Gamma \rangle \ ; \ (\mathtt{app} \times Id_D) \ ; \ \mathtt{ev}$. *By induction hypothesis,*

  $$\langle [t_1[y := u]]_\Gamma, [t_2[y := u]]_\Gamma \rangle = \langle \big(\langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma \rangle ; [t_1]_{\Gamma,y}\big), \big(\langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma \rangle ; [t_2]_{\Gamma,y}\big) \rangle$$

  which is equal to $\langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma \rangle \ ; \ \langle [t_1]_{\Gamma,y}, [t_2]_{\Gamma,y} \rangle$. *Hence,*

  $$[t_1 t_2[y := u]]_\Gamma = \langle \pi_1^k, \dots, \pi_k^k, [u]_\Gamma \rangle ; [t_1 t_2]_{\Gamma,y} \ .$$

- *If* $t = \{\!|\theta|\!\} \cdot u$, *it is similar to previous case.* □

Proposition (4.2.5). *If* $\mathscr{M} = (\mathbb{C}, D, \mathtt{lam}, \mathtt{app}, (\mathsf{c_i}^*)_{i=1}^n, \mathtt{case}, \grave{} \,)$ *is a* $\lambda_{\mathscr{C}}$*model, then for any* $\Gamma = \{x_1, \dots, x_k\}$ *and any terms* t_1, t_2 *such that* $\mathrm{fv}(t_1) \subseteq \Gamma$ *and* $t_1 \to t_2$, *the interpretation given in Fig. 4.2 satisfies* $[t_1]_\Gamma = [t_2]_\Gamma$.

Proof : *Let* t_1, t_2 *be two* $\lambda_{\mathscr{C}}$*-terms such that* $t_1 \to t_2$. *We prove by induction on the structure of* t_1 *that for any* Γ *containing all free variables of* t_1, $[t_1]_\Gamma = [t_2]_\Gamma$. *If the reduction does not involve a head redex, we immediately conclude with induction hypothesis. So we consider all possible reductions in head position:*

- $t_1 = (\lambda x.t)\, u$ *and* $t_2 = t[x := u]$.

  $[t_1]_\Gamma = \ D^k \xrightarrow{\langle (\Lambda(f_t);\mathtt{lam}), [u]_\Gamma \rangle} D \times D \xrightarrow{\mathtt{app} \times Id_D} D^D \times D \xrightarrow{\mathtt{ev}} D$

with $f_t = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t]_{\Gamma,x}} D$ *. Thus*

$$\begin{array}{rcll} [t_1]_\Gamma & = & \langle Id_D, [u]_\Gamma\rangle \;;\; (\Lambda(f_t); \mathtt{lam}; \mathtt{app}) \times Id_D \;;\; \mathtt{ev} & \\ & = & \langle Id_D, [u]_\Gamma\rangle \;;\; \Lambda(f_t) \times Id_D \;;\; \mathtt{ev} & (D1) \\ & = & \langle Id_D, [u]_\Gamma\rangle \;;\; f_t & (\textit{Def. of exponential}) \\ & = & [t[x := u]]_\Gamma & (\textit{Lem. 4.2.4}) \end{array}$$

- $t_1 = \lambda x.tx$ *(with* $x \notin \mathrm{fv}(t)$*) and* $t_2 = t$*. Then* $[t_1]_\Gamma = \Lambda(f_{tx}) \;;\; \mathtt{lam}$

  where $f_{tx} = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{\langle [t]_{\Gamma,x}, [x]_{\Gamma,x}\rangle} D \times D \xrightarrow{\mathtt{app}\times Id_D} D^D \times D \xrightarrow{\mathtt{ev}} D$.
  But $x \notin \mathrm{fv}(t)$ *implies* $[t]_{\Gamma,x} = \langle \pi_1^{k+1}, \ldots, \pi_k^{k+1}\rangle \;;\; [t]_\Gamma$ *by weakening property (Lem. 4.2.3), and* $[x]_{\Gamma,x} = \pi_{k+1}^{k+1}$.

  So $f_{tx} = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{\langle \langle \pi_1^{k+1}, \ldots, \pi_k^{k+1}\rangle, \pi_{k+1}^{k+1}\rangle} D^k \times D \xrightarrow{([t]_\Gamma; \mathtt{app})\times Id_D} D^D \times D \xrightarrow{\mathtt{ev}} D$.
  ($Id_{D^k \times D}$)

  By uniqueness of the exponential, $\Lambda(f_{tx}) = [t]_\Gamma; \mathtt{app}$*, and* $[t_1]_\Gamma = [t]_\Gamma; \mathtt{app}; \mathtt{lam} = [t]_\Gamma$ *by* $(D1)$.

- $t_1 = \{\!|\theta|\!\} \cdot \mathsf{c_i}$ *and* $t_2 = u_i$*, where* $\theta = \{\mathsf{c_j} \mapsto u_j / j \in J\}$*, with* $J \subseteq [\![1..n]\!]$.

  Then $[t_1]_\Gamma = \langle \langle f_1, \ldots, f_n\rangle, [\mathsf{c_i}]_\Gamma\rangle \;;\; \mathtt{case}$ *with* $f_j = \begin{cases} [u_j]_\Gamma & \text{if } j \in J \\ !_{D^k}; \iota & \text{otherwise} \end{cases}$
  and $[\mathsf{c_i}]_\Gamma = !_{D^k}; c_i^*$.

  The following diagram commutes:
  $$\begin{array}{ccccc} D^k & \xrightarrow{\langle \langle f_1, \ldots, f_n\rangle, !_{D^k}\rangle} & D^n \times \mathbf{1} & \xrightarrow{Id_{D^n} \times c_i^*} & D^n \times D \\ & \searrow^{\langle f_1, \ldots, f_n\rangle} & \cong\downarrow & (D2) & \downarrow{\scriptstyle \mathtt{case}} \\ & & D^n & \xrightarrow[\pi_i^n]{} & D \end{array}$$

  so $[t_1]_\Gamma \;=\; \langle f_1, \ldots, f_n\rangle \;;\; \pi_i^n \;=\; f_i \;=\; [u_i]_\Gamma$.

- $t_1 = \{\!|\theta|\!\} \cdot (tu)$ *and* $t_2 = (\{\!|\theta|\!\} \cdot t)\; u$.
  $[t_1]_\Gamma = \langle [\theta]_\Gamma, [tu]_\Gamma\rangle \;;\; \mathtt{case}$ *with* $[tu]_\Gamma = \langle [t]_\Gamma, [u]_\Gamma\rangle \;;\; (\mathtt{app} \times Id_D) \;;\; \mathtt{ev}$
  $[t_2]_\Gamma = \langle \big(\langle [\theta]_\Gamma, [t]_\Gamma\rangle \;;\; \mathtt{case}\big), [u]_\Gamma\rangle \;;\; (\mathtt{app} \times Id_D); \mathtt{ev}$
  So $[t_1]_\Gamma = [t_2]_\Gamma$ *because the following diagram commutes:*
  $$\begin{array}{ccccccc} & & D^n \times (D \times D) & \xrightarrow{Id \times (\mathtt{app} \times Id)} & D^n \times (D^D \times D) & \xrightarrow{Id \times \mathtt{ev}} & D^n \times D \\ & \nearrow^{\langle [\theta]_\Gamma, \langle [t]_\Gamma, [u]_\Gamma\rangle\rangle} & \cong\uparrow & (D3) & & & \searrow^{\mathtt{case}} \\ D^k & & & & & & D \\ & \searrow_{\langle \langle [\theta]_\Gamma, \langle [t]_\Gamma\rangle\rangle, [u]_\Gamma\rangle} & & & & & \nearrow_{\mathtt{ev}} \\ & & (D^n \times D) \times D & \xrightarrow[\mathtt{case} \times Id]{} & D \times D & \xrightarrow[\mathtt{app} \times Id]{} & D^D \times D \end{array}$$

- $t_1 = \{\!|\theta|\!\} \cdot \lambda x.t$ *and* $t_2 = \lambda x.\{\!|\theta|\!\} \cdot t$ *with* $x \notin \mathrm{fv}(\theta)$.

  $[t_1]_\Gamma \;= \langle [\theta]_\Gamma, (\Lambda(f_t); \mathtt{lam})\rangle \;;\; \mathtt{case}$ *with* $f_t = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t]_{\Gamma,x}} D$ *, and*

  $[t_2]_\Gamma \;= \Lambda(f_{\{\!|\theta|\!\}\cdot t}); \mathtt{lam}$ *with* $f_{\{\!|\theta|\!\}\cdot t} = \; D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{\langle [\theta]_{\Gamma,x}, [t]_{\Gamma,x}\rangle} D^n \times D \xrightarrow{\mathtt{case}} D$.

  So $\begin{array}{rcll} [t_1]_\Gamma & = & \langle [\theta]_\Gamma, (\Lambda(f_t); \mathtt{lam})\rangle \;;\; \mathtt{case} & \\ & = & \langle [\theta]_\Gamma, \Lambda(f_t)\rangle \;;\; (Id_{D^n} \times \mathtt{lam}) \;;\; \mathtt{case} & \\ & = & \langle [\theta]_\Gamma, \Lambda(f_t)\rangle \;;\; \mathtt{case}^\circ \;;\; \mathtt{lam} & \textit{by } (D4) \end{array}$

Hence $[t_1]_\Gamma = [t_2]_\Gamma$ *if* $\langle [\theta]_\Gamma, \Lambda(f_t)\rangle \;;\; \mathtt{case}^\circ = \Lambda(f_{\{\!|\theta|\!\}\cdot t})$.

Remember (page **??***) that* $\mathtt{case}^\circ = \Lambda(f_{\mathtt{case}})$*, with*

$$f_{\mathtt{case}} = \quad (D^n \times D^D) \times D \xrightarrow{\cong} D^n \times (D^D \times D) \xrightarrow{Id_{D^n} \times \mathtt{ev}} D^n \times D \xrightarrow{\mathtt{case}} D$$

Hence by Lem. 4.1.3, $[t_1]_\Gamma = [t_2]_\Gamma \quad$ *if* $\quad (\langle [\theta]_\Gamma, \Lambda(f_t)\rangle \times Id_d) \;;\; f_{\mathtt{case}} = f_{\{\!|\theta|\!\}\cdot t}$.

Remark that $(\langle [\theta]_\Gamma, \Lambda(f_t)\rangle \times Id_d) \;;\; f_{\mathtt{case}} = lhs \;;\; \mathtt{case}$*, with*

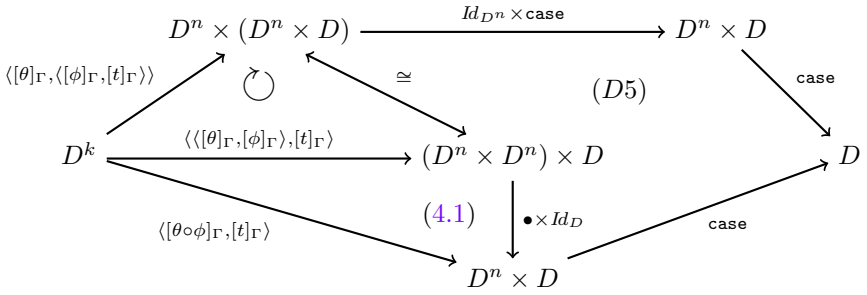
$$\begin{aligned} lhs \;&=\; D^k\!\times\! D \xrightarrow{\langle [\theta]_\Gamma, \Lambda(f_t)\rangle \times Id_D} (D^n\!\times\! D^D)\!\times\! D \xrightarrow{\cong} D^n\!\times\!(D^D\!\times\! D) \xrightarrow{Id_{D^n}\times \mathtt{ev}} D^n\!\times\! D \\ &=\; D^k\!\times\! D \xrightarrow{\langle (\pi_1\,;\,[\theta]_\Gamma), Id\rangle} D^n\!\times\!(D^k\!\times\! D) \xrightarrow{Id_{D^n}\times(\Lambda(f_t)\times Id_D)} D^n\!\times\!(D^D\!\times\! D) \xrightarrow{Id_{D^n}\times \mathtt{ev}} D^n\!\times\! D \\ &=\; D^k\!\times\! D \xrightarrow{\langle (\pi_1\,;\,[\theta]_\Gamma), Id\rangle} D^n\!\times\!(D^k\!\times\! D) \xrightarrow{Id_{D^n}\times f_t} D^n\!\times\! D \end{aligned}$$

On the other hand, $f_{\{\!|\theta|\!\}\cdot t} = rhs \;;\; \mathtt{case}$*, with*

$$\begin{aligned} rhs \;&=\; D^k\!\times\! D \xrightarrow{\cong} D^{k+1} \xrightarrow{\langle Id, Id\rangle} D^{k+1}\!\times\! D^{k+1} \xrightarrow{[\theta]_{\Gamma,x}\times[t]_{\Gamma,x}} D^n\!\times\! D \\ &=\; D^k\!\times\! D \xrightarrow{\cong} D^{k+1} \xrightarrow{\langle Id, Id\rangle} D^{k+1}\!\times\! D^{k+1} \xrightarrow{\langle \ldots, \pi_k^{k+1}\rangle \times Id} D^k\!\times\! D^{k+1} \xrightarrow{[\theta]_\Gamma\times[t]_{\Gamma,x}} D^n\!\times\! D \quad \textit{(Lem. 4.2.3)} \\ &=\; D^k\!\times\! D \xrightarrow{\langle Id, Id\rangle} (D^k\!\times\! D)\!\times\!(D^k\!\times\! D) \xrightarrow{\pi_1\times\cong} D^k\!\times\! D^{k+1} \xrightarrow{[\theta]_\Gamma\times[t]_{\Gamma,x}} D^n\!\times\! D \\ &=\; D^k\!\times\! D \xrightarrow{\langle Id, Id\rangle} (D^k\!\times\! D)\!\times\!(D^k\!\times\! D) \xrightarrow{(\pi_1\,;\,[\theta]_\Gamma)\times f_t} D^n\!\times\! D \end{aligned}$$

Finally $rhs = lhs = \langle (\pi_1\,;\,[\theta]_\Gamma)\;,\; f_t\rangle$*, and so* $[t_1]_\Gamma = [t_2]_\Gamma$.

- $t_1 = \{\!|\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t$ *and* $t_2 = \{\!|\theta\circ\phi|\!\}\cdot t$.
  $[t_1]_\Gamma \;= \big(\langle [\theta]_\Gamma, \langle [\phi]_\Gamma, [t]_\Gamma\rangle\rangle\big) \;;\; (Id_{D^n} \times \mathtt{case}) \;;\; \mathtt{case}$, *and*
  $[t_2]_\Gamma = \big(\langle [\theta\circ\phi]_\Gamma, [t]_\Gamma\rangle\big) \;;\; \mathtt{case}$.
  Both terms have the same interpretation if the following diagram commute:



The upper triangle commutes by uniqueness of the product, the triangle below commutes if $(D6)$ *commutes (consequence of Lem. 4.2.2), and the right part of the diagram is exactly* $(D5)$*. Also the interpretation is correct w.r.t.* CaseCase *if* $(D5)$ *and* $(D6)$ *commute.*

A.1.2 Proof of correctness of PER-model

The following lemma achieves the proof of Prop. 4.3.3.

Lemma. *In the category* $\mathbb{P}\mathrm{ER}_{\lambda_{\mathscr{C}}}$*, let* $D = \simeq_{\lambda_{\mathscr{C}}}$, $\mathtt{lam} = \mathtt{app} = Id_D$, $c^* = \overline{\lambda x.\mathsf{c}}^{\mathbf{1}\to D}$, $\mathtt{case} = \overline{t_{\mathtt{case}}}^{(D^n\times D)\to D}$*, with* $t_{\mathtt{case}} = \lambda x.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1 x))_{i=1}^n|\!\}\cdot\pi_2 x$*, and* $\grave{} = \overline{\lambda x.\{\!|\;|\!\}\cdot \mathsf{c}_1}^{\mathbf{1}\to D}$*. Then the diagrams* $(D2)$, $(D3)$, $(D5)$ *and* $(D6)$ *(of Fig. 4.1) commute.*

PROOF : *(D2): We show that* $rhs = \pi_i^n$, *where* $rhs = h_{\cong}$; $(Id_{D^n} \times \mathsf{c}_i^*)$; $\mathtt{case}$ *(with* $h_{\cong} = \overline{\lambda x.(\!|x,x|\!)}^{D^n \to D^n \times \mathbf{1}}$*). Remember that* $(Id_{D^n} \times \mathsf{c}_i^*) = \overline{\lambda x.(\!|\pi_1 x, (\lambda x.\mathsf{c}_i)(\pi_2 x)|\!)}$ *by Rem.* **??**. *We simplify rhs, considering terms up to* $\lambda_{\mathscr{C}}$*-equivalence (4.2).*

$$\begin{aligned}
rhs &= \overline{\lambda z.t_{\mathsf{case}}\big((\lambda x.(\!|\pi_1 x, (\lambda x.\mathsf{c}_i)x|\!))\ ((\lambda x.(\!|x,x|\!))z)\big)}^{D^n \to D} \\
&= \overline{\lambda z.t_{\mathsf{case}}\big((\!|\pi_1(\!|z,z|\!), (\lambda x.\mathsf{c}_i)(\pi_2(\!|z,z|\!))|\!)\big)}^{D^n \to D} \\
&= \overline{\lambda z.t_{\mathsf{case}}((\!|z,\mathsf{c}_i|\!))}^{D^n \to D} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\!|z,\mathsf{c}_i|\!)))_{i=1}^n|\!\} \cdot \pi_2(\!|z,\mathsf{c}_i|\!)}^{D^n \to D} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\!|z,\mathsf{c}_i|\!)))_{i=1}^n|\!\} \cdot \pi_2(\!|z,\mathsf{c}_i|\!)}^{D^n \to D} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n\ z)_{i=1}^n|\!\} \cdot \mathsf{c}_i}^{D^n \to D} \\
&= \overline{\lambda z.\pi_i^n\ z}^{D^n \to D} \qquad \textit{by } \textsc{CaseCons} \\
&= \pi_i^n
\end{aligned}$$

(D3): We show that $lhs = rhs$, *where* $lhs = (\mathtt{case} \times Id_D)$; $(\mathtt{app} \times Id_D)$; $\mathtt{ev}$, *and* $rhs = h_{\cong}$; $(Id_{D^n} \times (\mathtt{app} \times Id_D))$; $(Id_{D^n} \times \mathtt{ev})$; $\mathtt{case}$, *with*

$$h_{\cong} = \overline{\lambda x.(\!|\pi_1(\pi_1 x), (\!|\pi_2(\pi_1 x), \pi_2 x|\!)|\!)}^{(D^n \times D) \times D \to D^n \times (D \times D)}.$$

Notice that $\mathtt{app} \times Id_D = Id_{D \times D}$, *so* $lhs = (\mathtt{case} \times Id_D)$; $\mathtt{ev}$, *and*

$$rhs = h_{\cong}\ ;\ (Id_{D^n} \times \mathtt{ev}); \mathtt{case}.$$

$$\begin{aligned}
lhs &= \overline{\lambda z.(\lambda x.(\pi_1 x)(\pi_2 x))\ \big((\lambda x.(\!|t_{\mathsf{case}}(\pi_1 x), \pi_2 x|\!))z\big)} \\
&= \overline{\lambda z.(\lambda x.(\pi_1 x)(\pi_2 x))\ (\!|t_{\mathsf{case}}(\pi_1 z), \pi_2 z|\!)} \\
&= \overline{\lambda z.(t_{\mathsf{case}}(\pi_1 z))\ (\pi_2 z)} \\
&= \overline{\lambda z.\big(\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\pi_1 z)))_{i=1}^n|\!\} \cdot \pi_2(\pi_1 z)\big)(\pi_2 z)}
\end{aligned}$$

$$\begin{aligned}
rhs &= \overline{\lambda z.t_{\mathsf{case}}\ (\lambda y.(\!|\pi_1 y, (\lambda x.(\pi_1 x)(\pi_2 x))(\pi_2 y)|\!))\ ((\lambda x.(\!|\pi_1(\pi_1 x), (\!|\pi_2(\pi_1 x), \pi_2 x|\!)|\!))z)} \\
&= \overline{\lambda z.t_{\mathsf{case}}\ (\lambda y.(\!|\pi_1 y, (\pi_1(\pi_2 y))(\pi_2(\pi_2 y))|\!))\ (\!|\pi_1(\pi_1 z), (\!|\pi_2(\pi_1 z), \pi_2 z|\!)|\!)} \\
&= \overline{\lambda z.t_{\mathsf{case}}\ (\!|\pi_1(\pi_1 z), (\pi_2(\pi_1 z))(\pi_2 z)|\!)} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\pi_1 z)))_{i=1}^n|\!\} \cdot \big(\pi_2(\pi_1 z)\ (\pi_2 z)\big)} \\
&= \overline{\lambda z.\big(\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\pi_1 z)))_{i=1}^n|\!\} \cdot \pi_2(\pi_1 z)\big)(\pi_2 z)} \qquad \textit{by } \textsc{CaseApp}
\end{aligned}$$

(D5): Let $lhs = (\bullet \times Id_D)$; $\mathtt{case}$, *and* $rhs = h_{\cong}$; $(Id_{D^n} \times \mathtt{case})$; $\mathtt{case}$, *with*

$$h_{\cong} = \overline{\lambda x.(\!|\pi_1(\pi_1 x)\ ,\ (\!|\pi_2(\pi_1 x)\ ,\ \pi_2 x|\!)|\!)}^{(D^n \times D^n) \times D \to D^n \times (D^n \times D)}.$$

Then (D5) commutes means $lhs = rhs$.
Remember that $\bullet : D^n \times D^n \to D^n$ *is the pairing of all* $(Id_{D^n} \times \pi_i^n)$; $\mathtt{case}$. *Thus*

$$\begin{aligned}
\bullet &= \overline{\lambda x.(\!|\ldots, (\lambda y.t_{\mathsf{case}}\ (\!|\pi_1 y\ ,\ \pi_n^i(\pi_2 y)|\!))x, \ldots|\!)} \\
&= \overline{\lambda x.(\!|\ldots, t_{\mathsf{case}}\ (\!|\pi_1 x\ ,\ \pi_n^i(\pi_2 x)|\!), \ldots|\!)} \\
\bullet \times Id_D &= \overline{\lambda x.(\!|\ (\!|\ldots, t_{\mathsf{case}}\ (\!|\pi_1(\pi_1 x)\ ,\ \pi_n^i(\pi_2(\pi_1 x))|\!), \ldots|\!)\ ,\ \pi_2 x\ |\!)} \\
lhs &= \overline{\lambda z.t_{\mathsf{case}}\ (\!|\ (\!|\ldots, t_{\mathsf{case}}\ (\!|\pi_1(\pi_1 z)\ ,\ \pi_n^i(\pi_2(\pi_1 z))|\!), \ldots|\!)\ ,\ \pi_2 z\ |\!)} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto t_{\mathsf{case}}\ (\!|\pi_1(\pi_1 z)\ ,\ \pi_n^i(\pi_2(\pi_1 z))|\!))_{i=1}^n|\!\} \cdot \pi_2 z} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto t_{\mathsf{case}}\ (\!|\pi_1(\pi_1 z)\ ,\ \pi_n^i(\pi_2(\pi_1 z))|\!))_{i=1}^n|\!\} \cdot (\pi_2 z)} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \{\!|(\mathsf{c}_j \mapsto \pi_j^n(\pi_1(\pi_1 z)))_{j=1}^n|\!\} \cdot (\pi_n^i(\pi_2(\pi_1 z))))_{i=1}^n|\!\} \cdot (\pi_2 z)} \\
rhs &= \overline{\lambda z.t_{\mathsf{case}}\ \big((\lambda x.(\!|\pi_1 x\ ,\ t_{\mathsf{case}}(\pi_2 x)|\!))\ (\!|\pi_1(\pi_1 z)\ ,\ (\!|\pi_2(\pi_1 z)\ ,\ \pi_2 z|\!)|\!)\big)} \\
&= \overline{\lambda z.t_{\mathsf{case}}\ (\!|\pi_1(\pi_1 z)\ ,\ t_{\mathsf{case}}(\!|\pi_2(\pi_1 z)\ ,\ \pi_2 z|\!)|\!)} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\pi_1 z)))_{i=1}^n|\!\} \cdot t_{\mathsf{case}}\ (\!|\pi_2(\pi_1 z)\ ,\ \pi_2 z|\!)} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\pi_1 z)))_{i=1}^n|\!\} \cdot \{\!|(\mathsf{c}_j \mapsto \pi_j^n(\pi_2(\pi_1 z)))_{j=1}^n|\!\} \cdot (\pi_2 z)} \\
&= \overline{\lambda z.\{\!|(\mathsf{c}_j \mapsto \{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1(\pi_1 z)))_{i=1}^n|\!\} \cdot \pi_j^n(\pi_2(\pi_1 z)))_{j=1}^n|\!\} \cdot (\pi_2 z)} \quad \textit{(by } \textsc{CaseCase}\textit{)}
\end{aligned}$$

(*D*6): *This diagram commutes if lhs = rhs, with lhs* $= \pi_2 \ ; \ `$, *and rhs* $= (Id_{D^n} \times `) \ ; \ \mathsf{case}$.

$$\begin{array}{rcl} lhs & = & \overline{\lambda z.(\lambda x.\{\!|\}\!| \cdot \mathsf{c}_1)\ (\pi_2 z)}^{D^n \times \mathbf{1} \to D} \\ & = & \overline{\lambda z.\{\!|\}\!| \cdot \mathsf{c}_1}^{D^n \times \mathbf{1} \to D} \\ rhs & = & \overline{\lambda z.t_{\mathsf{case}}\ (\langle\!|\pi_1 z, \ , (\lambda x.\{\!|\}\!| \cdot \mathsf{c}_1)\ (\pi_2 z)|\!\rangle}^{D^n \times \mathbf{1} \to D} \\ & = & \overline{\lambda z.t_{\mathsf{case}}\ (\langle\!|\pi_1 z, \ , \{\!|\}\!| \cdot \mathsf{c}_1|\!\rangle}^{D^n \times \mathbf{1} \to D} \\ & = & \overline{\lambda z.\{\!|(\mathsf{c}_i \mapsto \pi_i^n(\pi_1 z))_{i=1}^n|\!\} \cdot \{\!|\}\!| \cdot \mathsf{c}_1}^{D^n \times \mathbf{1} \to D} \\ & = & \overline{\lambda z.\{\!|\}\!| \cdot \mathsf{c}_1}^{D^n \times \mathbf{1} \to D} \qquad (\textit{by } \textsc{CaseCase}) \end{array}$$

□

A.1.3 Proof of completeness of PER-model

In this section we give the proofs of all the lemmas of Sec. 4.3.4 that are needed for the completeness result (Theo. 4.2). All of them rely on rewriting arguments, although this thesis is not a thesis about rewriting.

Lemma (4.3.6). *$\lambda^-_{\mathscr{C}}$-reduction on completed terms*
Let t be a defined term. Then, for any term t',

$$\widetilde{t} \to_{\lambda^-_{\mathscr{C}}} t' \quad \textit{implies} \quad t' = \widetilde{t_0} \ \textit{ for some } t_0 \textit{ such that } t \to t_0.$$

(Remember that $\lambda^-_{\mathscr{C}}$ denotes the $\lambda_{\mathscr{C}}$-calculus without rule CaseCase).

Proof : *By structural induction on t. First notice that every* CaseCons *redex present in $\widetilde{t}$ corresponds to a* CaseCons *redex in t, as t is defined. Moreover, $\{\!|\ |\!\} \cdot \mathsf{c}_1$ is not reducible so every redex in a sub-term of $\widetilde{t}$ corresponds to a redex in a sub-term of t Also if the reduction $\widetilde{t} \to t'$ is performed in a (strict) sub-term of $\widetilde{t}$, we can immediately conclude with induction hypothesis. So it is sufficient to check the lemma for the five possible reductions in head position $\widetilde{t} \twoheadrightarrow t'$, which is trivial.* □

Lemma (4.3.7). CaseCase *reduction on completed terms*
For any term t, t',

$$\widetilde{t} \to_{cc} t' \quad \textit{implies} \quad t' \to^*_{cc} \widetilde{t_0} \ \textit{ for some } t_0 \textit{ such that } t \to_{cc} t_0$$

where $\to_{cc}$ denotes a reduction with rule CaseCase.

Proof : *By by structural induction on t. If the* CaseCase *reduction occurs in a strict sub-term of $\widetilde{t}$ then we conclude with induction hypothesis. Otherwise $t = \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot u$, and $t' = \{\!|\widetilde{\theta} \circ \widetilde{\phi}|\!\} \cdot \widetilde{u}$. Then we take $t_0 = \{\!|\theta \circ \phi|\!\} \cdot u$, since $\widetilde{\theta} \circ \widetilde{\phi} \to^*_{cc} \widetilde{\theta \circ \phi}$. Indeed, if $\phi = \{\mathsf{c}_i \mapsto u_i / i \in I\}$ then*

$$\begin{array}{rcl} \widetilde{\theta} \circ \widetilde{\phi} & = & \{\mathsf{c}_i \mapsto \{\!|\widetilde{\theta}|\!\} \cdot \widetilde{u_i} / i \in I\} \cup \{\mathsf{c}_i \mapsto \{\!|\widetilde{\theta}|\!\} \cdot \{\!|\}\!| \cdot \mathsf{c}_1 / i \notin I\} \\ \widetilde{\theta \circ \phi} & = & \{\mathsf{c}_i \mapsto \{\!|\widetilde{\theta}|\!\} \cdot \widetilde{u_i} / i \in I\} \cup \{\mathsf{c}_i \mapsto \{\!|\}\!| \cdot \mathsf{c}_1 / i \notin I\} \end{array}$$

Also $t' \to^*_{cc} \widetilde{t_0}$. □

Lemma (4.3.9). *Commutation case-completion/*CC*-normal form*
For any term t,

$$\Downarrow (\widetilde{t}) = \widetilde{\Downarrow t} \ .$$

PROOF : *By induction on the size of the maximal reduction* $\widetilde{t} \to_{cc} \Downarrow (\widetilde{t})$. *If* $\widetilde{t} = \Downarrow (\widetilde{t})$, *then* $\widetilde{t}$ *is* CASECASE*-normal, and so is* t *(Fact.4.3.4). Thus* $t = \Downarrow t$ *and* $\widetilde{t} = \widetilde{\Downarrow t}$. *Otherwise let* $\widetilde{t} \to_{cc} t' \to_{cc}^* \Downarrow (\widetilde{t})$. *By Lem. 4.3.7, there is a term* t_0 *such that* $t' \to_{cc}^* \widetilde{t_0}$ *and* $t \to_{cc} t_0$. *Hence* $\widetilde{t} \to_{cc}^+ \widetilde{t_0} \to_{cc}^* \Downarrow (\widetilde{t}) = \Downarrow (\widetilde{t_0})$. *By induction hypothesis,* $\Downarrow (\widetilde{t_0}) = \widetilde{\Downarrow t_0}$. *Moreover* $\Downarrow t_0 = \Downarrow t$, *so* $\widetilde{(\Downarrow t)} = \widetilde{(\Downarrow t_0)} = \Downarrow (\widetilde{t_0}) = \Downarrow (\widetilde{t})$. □

Lemma (4.3.10)**.** *For any terms* t, t', *if* $t \to_{\lambda_{\mathscr{C}}^-} t'$ *then there exists a term* u *such that*

$$\Downarrow t \to_{\lambda_{\mathscr{C}}^-}^* u \to_{cc}^* \Downarrow t' \ .$$

PROOF : *The proof proceeds by induction on* $s(t)$, *the structural measure of* t *(Def. 2.1.2). For any term* s *(or any case-binding* θ*),* s' *(resp.* θ'*) represents a term (resp. a case-binding) such that* $s \to_{\lambda_{\mathscr{C}}} s'$ *(resp.* $\theta_{\mathsf{c}} \to_{\lambda_{\mathscr{C}}} \theta'_{\mathsf{c}}$ *for some* $\mathsf{c} \in \mathrm{dom}(\theta)$, *and* $\theta_{\mathsf{c}'} = \theta'_{\mathsf{c}'}$ *for* $\mathsf{c}' \neq \mathsf{c}$*)*

- *If* t *is an application, either* $t = t_1 t_2$ *and* $t' = t'_1 t_2$ *(or* $t' = t_1 t'_2$*) and we conclude with induction hypotheses, or* $t = (\lambda x.t_1)t_2$ *and* $t' = t_1[x := t_2]$. *In that case,* $\Downarrow t = (\lambda x.\ \Downarrow t_1)\ \Downarrow t_2 \to_{\lambda_{\mathscr{C}}^-} (\Downarrow t_1)[x :=\Downarrow t_2] \to_{cc}^* \Downarrow (\Downarrow t_1)[x :=\Downarrow t_2]$. *Moreover,* $\Downarrow (\Downarrow t_1)[x :=\Downarrow t_2] = \Downarrow (t_1[x := t_2])$. *Thus* $\Downarrow t \to_{\lambda_{\mathscr{C}}^-} (\Downarrow t_1)[x :=\Downarrow t_2] \to_{cc}^* \Downarrow t'$.
- *If* t *is an abstraction, either* $t = \lambda x.t_0$ *and* $t' = \lambda x.t'_0$ *and we conclude with induction hypothesis, or* $t = \lambda x.t'x$ *with* $x \notin \mathrm{fv}(t')$. *In that case,* $\Downarrow t = \lambda x.\ \Downarrow t'x \to_{\lambda_{\mathscr{C}}^-} \Downarrow t'$.
- *If* $t = \{\!|\theta|\!\} \cdot x$, *then* $t' = \{\!|\theta'|\!\} \cdot x$ *and we conclude with induction hypothesis.*
- *If* $t = \{\!|\theta|\!\} \cdot \mathsf{c}$, *then either* $t' = \{\!|\theta'|\!\} \cdot \mathsf{c}$ *and we conclude with induction hypothesis, or* $t' = \theta_{\mathsf{c}}$ *and* $\Downarrow t = \{\!|\ \Downarrow \theta|\!\} \cdot \mathsf{c} \to_{\lambda_{\mathscr{C}}^-} \Downarrow \theta_{\mathsf{c}}$.
- *If* $t = \{\!|\theta|\!\} \cdot t_1 t_2$, *then either* $t' = \{\!|\theta'|\!\} \cdot t_1 t_2$ *and we conclude with induction hypothesis, or* $t' = \{\!|\theta|\!\} \cdot t_0$ *with* $t_1 t_2 \to_{\lambda_{\mathscr{C}}^-} t_0$ *or* $t' = (\{\!|\theta|\!\} \cdot t_1)t_2$.

  In the second case, by induction hypothesis there is some u_0 *such that* $\Downarrow t_1 t_2 \to_{\lambda_{\mathscr{C}}^-}^* u_0 \to_{cc}^* \Downarrow t_0$. *Hence*

  $$\Downarrow t = \{\!|\ \Downarrow \theta|\!\} \cdot \Downarrow t_1 t_2 \to_{\lambda_{\mathscr{C}}^-}^* \{\!|\ \Downarrow \theta|\!\} \cdot u_0 \to_{cc}^* \{\!|\ \Downarrow \theta|\!\} \cdot \Downarrow t_0 \to_{cc}^* \Downarrow \{\!|\ \Downarrow \theta|\!\} \cdot \Downarrow t_0 \ .$$

  Moreover, every sub-term of $\Downarrow\ t'$ *is in* CASECASE *normal form, so* $\Downarrow t' = \Downarrow \{\!|\ \Downarrow \theta|\!\} \cdot \Downarrow t_0$. *Thus* $\Downarrow t \to_{\lambda_{\mathscr{C}}^-}^* \{\!|\ \Downarrow \theta|\!\} \cdot u_0 \to_{cc}^* \Downarrow t'$.

  In the last case, $\Downarrow t = \{\!|\ \Downarrow \theta|\!\} \cdot (\Downarrow t_1\ \Downarrow t_2)$, *so*

  $$\Downarrow t \to_{\lambda_{\mathscr{C}}^-} (\{\!|\ \Downarrow \theta|\!\} \cdot \Downarrow t_1)\ \Downarrow t_2 \to_{cc}^* \Downarrow (\{\!|\ \Downarrow \theta|\!\} \cdot \Downarrow t_1)\ \Downarrow t_2 = \Downarrow \{\!|\theta|\!\} \cdot t_1\ \Downarrow t_2.$$

- *If* $t = \{\!|\theta|\!\} \cdot \lambda x.t_0$, *idem as previous case.*
- *If* $t = \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t_0$, *then either* $t' = \{\!|\theta|\!\} \cdot \{\!|\phi'|\!\} \cdot t_0$, *or* $t' = \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t'_0$, *or* $t' = \{\!|\theta'|\!\} \cdot \{\!|\phi|\!\} \cdot t_0$.

  In the first case, write $t_1 = \{\!|\theta \circ \phi|\!\} \cdot t_0$ *and* $t'_1 = \{\!|\theta \circ \phi'|\!\} \cdot t_0$. *Remark that* $s(t_1) < s(t)$ *(since the structural measure decreases by* CASECASE*-reduction), and that* $t_1 \to_{\lambda_{\mathscr{C}}} t'_1$. *By induction hypothesis, there is some* u *such that* $\Downarrow t_1 \to_{\lambda_{\mathscr{C}}^-}^* u \to_{cc}^* \Downarrow t'_1$. *Since* $\Downarrow t = \Downarrow t_1$ *and* $\Downarrow t' = \Downarrow t'_1$ *we are done.*

  In the second case, same method but with $t'_1 = \{\!|\theta \circ \phi|\!\} \cdot t'_0$.

In the last case, write $t = \{\!|\theta|\!\} \cdot \{\!|\phi_1|\!\} \cdot \cdots \cdot \{\!|\phi_k|\!\} \cdot u_0$, *where* u_0 *is not a case construct (thus* $k \geq 1$*). Then* $\Downarrow t = \{\!|\ \Downarrow (\theta \circ \psi)|\!\} \cdot \Downarrow u_0$, *with* $\psi = \phi_1 \circ (\cdots \circ \phi_k)$, *and* $\Downarrow t' = \{\!|\ \Downarrow (\theta' \circ \psi)|\!\} \cdot \Downarrow u_0$ *(since* $((\theta \circ \phi_1) \circ \cdots) \circ \phi_k \to_{cc}^* \theta \circ \psi$*).*

Let us explicit $\Downarrow t$ *and* $\Downarrow t'$*:*

$$\begin{array}{rcl} \Downarrow t & = & \{\!|\mathsf{c} \mapsto \Downarrow \{\!|\theta|\!\} \cdot \psi_\mathsf{c} \,/\, \mathsf{c} \in \mathrm{dom}(\psi)|\!\} \cdot \Downarrow u_0 \\ \Downarrow t' & = & \{\!|\mathsf{c} \mapsto \Downarrow \{\!|\theta'|\!\} \cdot \psi_\mathsf{c} \,/\, \mathsf{c} \in \mathrm{dom}(\psi)|\!\} \cdot \Downarrow u_0 \end{array}$$

Remark that $s(\{\!|\theta|\!\} \cdot \psi_\mathsf{c}) \leq s(t)$ *(the structural measure decreases by* CaseCase*-reduction, and preserves the order of sub-term relation), and that* $\{\!|\theta|\!\} \cdot \psi_\mathsf{c} \to_{\lambda^-_{\mathscr{C}}} \{\!|\theta'|\!\} \cdot \psi_\mathsf{c}$*. Hence , by induction hypothesis, for each* $\mathsf{c} \in \mathrm{dom}(\psi)$ *there is a term* u_c *such that* $\Downarrow \{\!|\theta|\!\} \cdot \psi_\mathsf{c} \to_{\lambda^-_{\mathscr{C}}}^* u_\mathsf{c} \to_{cc}^* \Downarrow \{\!|\theta'|\!\} \cdot \psi_\mathsf{c}$*. Thus*

$$\Downarrow t \to_{\lambda^-_{\mathscr{C}}}^* u \to_{cc}^* \Downarrow t' \qquad \textit{for} \qquad u = \{\!|\mathsf{c} \mapsto u_\mathsf{c} \,/\, \mathsf{c} \in \mathrm{dom}(\psi)|\!\} \cdot \Downarrow u_0 \ . \qquad \square$$

A.2 Abstract machine and CPS translation

A.2.1 Abstract machine correction

Proposition (5.1.1)**.** *If a* $\lambda_{\mathscr{C}}$*-term* t *has a head normal form* h*, then* $eval(t) = h$*.*

Proof : *By induction on the reduction* $t \twoheadrightarrow_{\mathrm{w}}^* h$*. If* t *itself is in normal form, it follows from the definition. Now we prove that* $t \twoheadrightarrow_{\mathrm{w}} t'$ *implies* $eval(t) = eval(t')$*, by structural induction on* t*. Write* $t = t_0 \dots t_k$*, where* t_0 *is not an application. Three cases can occur:*

1. $t_0 = \lambda x.u$ *and* $k \leq 1$ *and* $t' = u[x := t_1]t_2 \dots t_k$*.*
   Then both `load` (t) *and* `load` (t') *reach state* $\diamond \ \star \ u[x := t_1] \ \star \ t_2 \dots t_k$*. Hence* $eval(t) = eval(t')$*.*
2. $t_0 = \maltese$ *and* $k \leq 1$ *and* $t' = \maltese\, t_2 \dots t_k$*. Then* $eval(t) = eval(t') = \maltese$*.*
3. $t_0 \twoheadrightarrow_{\mathrm{w}} t'_0$ *and* $t' = t'_0 t_1 \dots t_k$
   - *If* $k = 0$*, five different cases may arise:*
   - $t = \{\!|\theta|\!\} \cdot \lambda x.u$ *and* $t' = \lambda x.\{\!|\theta|\!\} \cdot u$
   - $t = \{\!|\theta|\!\} \cdot \maltese$ *and* $t' = \maltese$

   In both cases $eval(t) = t'$
   - $t = \{\!|\theta|\!\} \cdot \mathsf{c}$ *and* $\mathsf{c} \mapsto t' \in \theta$
   - $t = \{\!|\theta|\!\} \cdot u_1 u_2$ *and* $t' = (\{\!|\theta|\!\} \cdot u_1)u_2$
   - $t = \{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot u$ *and* $t' = \{\!|\theta \circ \phi|\!\} \cdot u$

   In those three cases, `load` (t) *and* `load` (t') *both reach a same state (resp.* $(\diamond \star t' \star \diamond)$ *and* $(\theta \ \star \ u_1 \ \star \ u_2 \cdot \diamond)$ *and* $(\theta \circ \phi \ \star \ u \ \star \ \diamond)$*). Also* $eval(t) = eval(t')$*.*
   - *If* $k \geq 1$*, by induction hypothesis* $eval(t_0) = eval(t'_0)$*. Moreover, writing* $\blacktriangleright_{\setminus\maltese}$ *evaluation steps that are not* $\langle\theta\rangle \ \star \ \maltese \ \star \ \pi \ \blacktriangleright \ \diamond \ \star \ \maltese \ \star \ \diamond$*, we can remark that*

$$\begin{array}{c} \langle\theta\rangle \ \star \ t \ \star \ \pi_1 \quad \blacktriangleright_{\setminus\maltese}^* \quad \langle\theta\rangle' \ \star \ t' \ \star \ \pi'_1 \\ \textit{implies} \\ \langle\theta\rangle \ \star \ t \ \star \ \pi_1\pi_2 \quad \blacktriangleright_{\setminus\maltese}^* \quad \langle\theta\rangle' \ \star \ t' \ \star \ \pi'_1\pi_2 \end{array} \tag{A.1}$$

   If $eval(t_0) \neq \maltese$*, then*

$$\begin{array}{llll} \mathtt{load}(t_0) & = & \diamond \ \star \ t_0 \ \star \ \diamond & \mathtt{load}(t'_0) \quad = \quad \diamond \ \star \ t'_0 \ \star \ \diamond \\ \mathtt{load}(t_0) & \blacktriangleright_{\setminus\maltese}^* & \langle\theta\rangle \ \star \ u_0 \ \star \ \pi_0 & \mathtt{load}(t'_0) \quad \blacktriangleright_{\setminus\maltese}^* \quad \langle\theta\rangle' \ \star \ u'_0 \ \star \ \pi'_0 \end{array}$$

where $\langle\theta\rangle \star u_0 \star \pi_0$ *(written* s_0*) and* $\langle\theta\rangle' \star u_0' \star \pi_0'$ *(written* s_0'*) are final and* $\mathtt{unload}(s_0) = \mathtt{unload}(s_0') = eval(t_0)$.
Write $\pi = t_1 \cdots t_k \cdot \diamond$. *Hence* (**??**) *implies*

$$\begin{array}{lllllllllllll} \mathtt{load}(t) & \blacktriangleright^*_{\setminus\maltese} & \diamond & \star & t_0 & \star & \pi & \blacktriangleright^*_{\setminus\maltese} & \langle\theta\rangle & \star & u_0 & \star & \pi_0\pi \quad (= s) \\ \mathtt{load}(t') & \blacktriangleright^*_{\setminus\maltese} & \diamond & \star & t_0' & \star & \pi & \blacktriangleright^*_{\setminus\maltese} & \langle\theta\rangle' & \star & u_0' & \star & \pi_0'\pi \quad (= s') \end{array}$$

- *If* u_0 *is not an abstraction, then* s_0 *is final and* $\mathtt{unload}(s_0) = \mathtt{unload}(s_0')$ *implies* $s_0 = s_0'$ *so* $s = s'$.
- *If* u_0 *is an abstraction, then either* $s_0 = s_0'$ *and we are done, or* $s_0 = \theta \star \lambda x.u_1 \star \diamond$ *and* $s_0' = \diamond \star \lambda x.\{\!|\theta|\!\} \cdot u_1 \star \diamond$ *(or conversely) with* $x \notin \mathrm{fv}(\theta)$. *Then we can conclude like in case 1 (*s *and* s' *produce the same state).*

If $eval(t_0) = \maltese$, *then*

$$\mathtt{load}(t_0) = \diamond \star t_0 \star \diamond \blacktriangleright^*_{\setminus\maltese} \langle\theta\rangle \star \maltese \star \pi_0 \blacktriangleright \diamond \star \maltese \star \diamond$$

Writing $\pi = t_1 \cdots t_k \cdot \diamond$, (**??**) *implies*

$$\mathtt{load}(t) \blacktriangleright^*_{\setminus\maltese} \diamond \star t_0 \star \pi \blacktriangleright^*_{\setminus\maltese} \langle\theta\rangle \star \maltese \star \pi_0\pi \blacktriangleright \diamond \star \maltese \star \diamond$$

Also $eval(t) = \maltese$. *The same reasoning on* t' *leads to* $eval(t') = \maltese$. □

A.2.2 From continuation model to $\lambda_{\mathscr{C}}$-model

Lemma (5.3.6). *Let* $M_{\cong} = \langle\!|\pi_{11}(x), \langle\!|\pi_{21}(x), \pi_2(x)|\!\rangle|\!\rangle$.
The typing judgement $x : (D^n \times D) \times D \vdash_p M_{\cong} : D^n \times (D \times D)$ *is derivable.*
Write $h_{\cong} = \lfloor M_{\cong} \rfloor_{x:(D^n\times D)\times D}$. *Then,*

$$(\mathtt{case} \times Id_D); (\mathtt{app} \times Id_D); \mathtt{ev} \;=\; h_{\cong}; (Id_{D^n} \times (\mathtt{app} \times Id_D)); (Id_{D^n} \times \mathtt{ev}); \mathtt{case}\ .$$

Proof : *On the one hand, let* $f = (\mathtt{case} \times Id_D); (\mathtt{app} \times Id_D); \mathtt{ev}$.
Then $f = \big((\mathtt{case}; \mathtt{app}) \times Id_D\big); \mathtt{ev} = \langle(\pi_1; \mathtt{case}; \mathtt{app}), \pi_2\rangle; \mathtt{ev}$.
By (B.2), $\lceil \mathtt{case}; \mathtt{app} \rceil \to_p^* \lambda z.M_{\mathtt{app}}[z := M_{\mathtt{case}}]$. *Hence*

$$\begin{array}{rcl} \lceil \mathtt{case}; \mathtt{app} \rceil & \to_p^* & \lambda z.\lambda x k.\ let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ M_{\mathtt{case}}\ \big(\downarrow_c \langle\!|x_\theta, \downarrow_s \langle\!|x, x_\pi|\!\rangle|\!\rangle\big) \\ & \to_p^* & \lambda z.\lambda x k.\ let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ let\ \langle\!|y_\theta, y_\pi|\!\rangle := \langle\!|x_\theta, \downarrow_s \langle\!|x, x_\pi|\!\rangle|\!\rangle\ in \\ & & \quad let\ \langle\!|y_\phi, y|\!\rangle := z\ in\ y\ \big(\downarrow_c \langle\!|\langle\!|M_1, \ldots, M_n|\!\rangle_n, y_\pi|\!\rangle\big)\ , \\ \textit{where } M_i & = & \lambda k'.let\ \langle\!|z_\theta, z_\pi|\!\rangle :=\uparrow_c k'\ in \\ & & \quad let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := y_\phi\ in\ x_i\ \big(\downarrow_c \langle\!|y_\theta, z_\pi|\!\rangle\big) \\ \lceil \mathtt{case}; \mathtt{app} \rceil & \to_p^* & \lambda z x k.\ let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ let\ \langle\!|y_\phi, y|\!\rangle := z\ in \\ & & \quad y\ \big(\downarrow_c \langle\!|\langle\!|M_1', \ldots, M_n'|\!\rangle_n, \downarrow_s \langle\!|x, x_\pi|\!\rangle|\!\rangle\big)\ , \\ \textit{where } M_i' & = & \lambda k'.let\ \langle\!|z_\theta, z_\pi|\!\rangle :=\uparrow_c k'\ in \\ & & \quad let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := y_\phi\ in\ x_i\ \big(\downarrow_c \langle\!|x_\theta, z_\pi|\!\rangle\big) \end{array}$$

Moreover $\dot{\pi_1} \to_p \lambda z.\pi_1(z)$, *and so by* (B.2),

$$\lceil \pi_1; \dot{\mathtt{case}}; \mathtt{app} \rceil \to_p^* \lambda z.\lambda x k.\ let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ let\ \langle\!|y_\phi, y|\!\rangle := \pi_1(z)\ in \quad .$$
$$y\ \big(\downarrow_c \langle\!|\langle\!|M_1', \ldots, M_n'|\!\rangle_n, \downarrow_s \langle\!|x, x_\pi|\!\rangle|\!\rangle\big)$$

In the same way, $\dot{\pi_1} \to_p \lambda z.\pi_1(z)$ *and so by* (B.3),

$$\begin{array}{rll}
\dot f & \to_p^* & \lambda z.\ (\ \lambda xk.\ let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ let\ \langle\!| y_\phi, y |\!\rangle := \pi_1(z)\ in \\
& & \qquad y\ (\downarrow_c \langle\!| \langle\!| M'_1, \dots, M'_n |\!\rangle_n, \downarrow_s \langle\!| x, x_\pi |\!\rangle |\!\rangle)\ )\quad \pi_2(z) \\
& \to_p^* & \lambda z.\lambda k.\ \ let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ let\ \langle\!| y_\phi, y |\!\rangle := \pi_1(z)\ in \\
& & \qquad y\ (\downarrow_c \langle\!| \langle\!| M'_1, \dots, M'_n |\!\rangle_n, \downarrow_s \langle\!| \pi_2(z), x_\pi |\!\rangle |\!\rangle) \\
& \to_p^* & \lambda zk.\ let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ \ \pi_{21}(z)\ (\downarrow_c \langle\!| \langle\!| N_1, \dots, N_n |\!\rangle_n, \downarrow_s \langle\!| \pi_2(z), x_\pi |\!\rangle |\!\rangle)
\end{array}$$
$$with\ N_i\ =\ \lambda k'.let\ \langle\!| z_\theta, z_\pi |\!\rangle :=\uparrow_c k'\ in\ let\ \langle\!| x_1; \dots; x_n |\!\rangle_n := \pi_{11}(z)\ in\ x_i\ (\downarrow_c \langle\!| x_\theta, z_\pi |\!\rangle)$$

On the other hand, let $g = h_\cong ; (Id_{D^n} \times (\mathsf{app} \times Id_D)) ; (Id_{D^n} \times \mathsf{ev}) ; \mathsf{case}$.
Then $g = h_\cong ; \big(Id_{D^n} \times (\langle (\pi_1 ; \mathsf{app})\ ,\ \pi_2 \rangle\ ;\ \mathsf{ev})\big) ; \mathsf{case}$. *Then*

$$\begin{array}{rll}
\lceil \langle (\pi_1 ; \mathsf{app})\ ,\ \pi_2 \rangle\ ;\ \mathsf{ev} \rceil & \to_p^* & \lambda x.\ M_{\mathsf{app}}[z := \pi_1(x)]\ \pi_2(x) \qquad (B.3) \\
\lceil Id_{D^n} \times (\langle (\pi_1 ; \mathsf{app})\ ,\ \pi_2 \rangle\ ;\ \mathsf{ev}) \rceil & \to_p^* & \lambda y.\ \langle\!| \pi_1(y), M_{\mathsf{app}}[z := \pi_{12}(y)]\ \pi_{22}(y) |\!\rangle \qquad (B.7) \\
h_\cong ; \big(Id_{D^n} \times (\langle (\pi_1 ; \mathsf{app})\ ,\ \pi_2 \rangle\ ;\ \mathsf{ev}\big) & & \\
& \to_p^* & \lambda x.\ \langle\!| \pi_1(M_\cong), M_{\mathsf{app}}[z := \pi_{12}(M_\cong)]\ \pi_{22}(M_\cong) |\!\rangle \qquad (B.2) \\
& \to_p^* & \lambda x.\ \langle\!| \pi_{11}(x), M_{\mathsf{app}}[z := \pi_{21}(x)]\ \pi_2(x) |\!\rangle
\end{array}$$
Still by (B.2), $\dot g \to_p^* \lambda x.M_{\mathsf{case}}[z := \langle\!| \pi_{11}(x), M_{\mathsf{app}}[z := \pi_{21}(x)]\ \pi_2(x) |\!\rangle]$, *which means*

$$\begin{array}{l}
\dot g \to_p^* \ \lambda x.\lambda k.let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ let\ \langle\!| y_\phi, y |\!\rangle := \langle\!| \pi_{11}(x), M_{\mathsf{app}}[z := \pi_{21}(x)]\ \pi_2(x) |\!\rangle\ in \\
\qquad\qquad y\ (\downarrow_c \langle\!| \langle\!| M_1, \dots, M_n |\!\rangle_n, x_\pi |\!\rangle)
\end{array}$$
$$with\ M_i = \lambda k'.let\ \langle\!| z_\theta, z_\pi |\!\rangle :=\uparrow_c k'\ in\ let\ \langle\!| x_1; \dots; x_n |\!\rangle_n := y_\phi\ in\ x_i\ (\downarrow_c \langle\!| x_\theta, z_\pi |\!\rangle).$$
$$\begin{array}{l}
\dot g \to_p^* \ \lambda x.\lambda k.let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in \\
\qquad\qquad M_{\mathsf{app}}[z := \pi_{21}(x)]\ \pi_2(x)\ (\downarrow_c \langle\!| \langle\!| N_1, \dots, N_n |\!\rangle_n, x_\pi |\!\rangle)
\end{array}$$
$$with\ N_i = \lambda k'.let\ \langle\!| z_\theta, z_\pi |\!\rangle :=\uparrow_c k'\ in\ let\ \langle\!| x_1; \dots; x_n |\!\rangle_n := \pi_{11}(x)\ in\ x_i\ (\downarrow_c \langle\!| x_\theta, z_\pi |\!\rangle).$$

Notice that $M_{\mathsf{app}}[z := \pi_{21}(x)]\ \pi_2(x)\ \to_p^*$
$$\lambda k'.let\ \langle\!| y_\theta, y_\pi |\!\rangle :=\uparrow_c k'\ in\ \pi_{21}(x)\ (\downarrow_c \langle\!| y_\theta, \downarrow_s \langle\!| \pi_2(x), y_\pi |\!\rangle |\!\rangle).$$
Hence $\begin{array}{rll} \dot g \to_p^* & \lambda x.\lambda k.let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ let\ \langle\!| y_\theta, y_\pi |\!\rangle := \langle\!| \langle\!| N_1, \dots, N_n |\!\rangle_n, x_\pi |\!\rangle\ in & . \\ & \qquad \pi_{21}(x)\ (\downarrow_c \langle\!| y_\theta, \downarrow_s \langle\!| \pi_2(x), y_\pi |\!\rangle |\!\rangle) & \\ \to_p^* & \lambda x.\lambda k.let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ \pi_{21}(x)\ (\downarrow_c \langle\!| \langle\!| N_1, \dots, N_n |\!\rangle_n, \downarrow_s \langle\!| \pi_2(x), x_\pi |\!\rangle |\!\rangle) & \end{array}$

Finally $\dot f$ *and* $\dot g$ *reduce on the same term (up to* α*-equivalence), thus* $f = g$ *by* (B.1). □

Lemma (5.3.7). *Write* $M_\cong = \langle\!| \pi_{11}(z), \langle\!| \pi_{21}(z), \pi_2(z) |\!\rangle |\!\rangle$,
and $h_\cong = \lfloor M_{eq} \rfloor_{z:(D^n \times D^n) \times D}$. *It is the commutation isomorphism from* $(D^n \times D^n) \times D$ *to* $D^n \times (D^n \times D)$. *Then,*

$$(\bullet \times Id_D)\ ;\ \mathsf{case}\ \ =\ \ h_\cong\ ;\ (Id_{D^n} \times \mathsf{case})\ ;\ \mathsf{case}\ .$$

PROOF : *Let* $f = (\bullet \times Id_D)\ ;\ \mathsf{case}$. *Remember that* $\bullet = \langle \dots, ((Id_{D^n} \times \pi_i^n) ; \mathsf{case}), \dots \rangle$.
Hence,

$$\begin{array}{rll}
\lceil Id_{D^n} \times \pi_i^n \rceil & \to_p^* & \lambda z.\langle\!| \pi_1(z), \pi_i^n(\pi_2(z)) |\!\rangle \qquad (B.7) \\
\lceil (Id_{D^n} \times \pi_i^n)\ ;\ \mathsf{case} \rceil & \to_p^* & \lambda z.M_{\mathsf{case}}[z := \langle\!| \pi_1(z), \pi_i^n(\pi_2(z)) |\!\rangle] \qquad (B.2) \\
\bullet & \to_p^* & \lambda z.M_\bullet \qquad (B.5) \\
\multicolumn{3}{l}{\quad with\ M_\bullet = \langle\!| \dots, M_{\mathsf{case}}[z := \langle\!| \pi_1(z)\ ,\ \pi_i^n(\pi_2(z)) |\!\rangle], \dots |\!\rangle_n} \\
\lceil \bullet \times Id_D \rceil & \to_p^* & \lambda z.\langle\!| M_\bullet[z := \pi_1(z)]\ ,\ \pi_2(z) |\!\rangle \qquad (B.7) \\
\dot f & \to_p^* & \lambda z.M_{\mathsf{case}}[z := \langle\!| M_\bullet[z := \pi_1(z)]\ ,\ \pi_2(z) |\!\rangle] \qquad (B.2)
\end{array}$$

Remember that
$$M_{\mathsf{case}} = \lambda k.let\ \langle\!| x_\theta, x_\pi |\!\rangle :=\uparrow_c k\ in\ let\ \langle\!| y_\phi, y |\!\rangle := z\ in\ y\ (\downarrow_c \langle\!| \langle\!| M_1, \dots, M_n |\!\rangle_n, x_\pi |\!\rangle)$$
$$where\ M_i\ =\ \lambda k'.let\ \langle\!| z_\theta, z_\pi |\!\rangle :=\uparrow_c k'\ in\ let\ \langle\!| x_1; \dots; x_n |\!\rangle_n := y_\phi\ in\ x_i\ (\downarrow_c \langle\!| x_\theta, z_\pi |\!\rangle)\ .$$

Hence

$$\begin{array}{rcl}
\dot{f} & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi\rangle := \uparrow_c k\ in\ let\ \langle y_\phi, y\rangle := \langle M_\bullet[z := \pi_1(z)]\ ,\ \pi_2(z)\rangle\ in \\
& & \qquad y\ (\downarrow_c \langle\langle M_1, \ldots, M_n\rangle_n, x_\pi\rangle) \\
\dot{f} & \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi\rangle := \uparrow_c k\ in\ \pi_2(z)\ (\downarrow_c \langle\langle M'_1, \ldots, M'_n\rangle_n, x_\pi\rangle) \\
\multicolumn{3}{l}{where\ M'_i =\quad \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in\quad let\ \langle x_1; \ldots; x_n\rangle_n := M_\bullet[z := \pi_1(z)]\ in} \\
\multicolumn{3}{r}{x_i\ (\downarrow_c \langle x_\theta, z_\pi\rangle)} \\
& \to_p^* & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in\ M_{\mathsf{case}}[z := \langle \pi_{11}(z), \pi_i^n(\pi_{21}(z))\rangle]\ (\downarrow_c \langle x_\theta, z_\pi\rangle) \\
& \to_p^* & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in\ let\ \langle x'_\theta, x'_\pi\rangle := \langle x_\theta, z_\pi\rangle\ in \\
& & \qquad let\ \langle y_\phi, y\rangle := \langle \pi_{11}(z), \pi_i^n(\pi_{21}(z))\rangle\ in\ y\ (\downarrow_c \langle\langle N_1, \ldots, N_n\rangle_n, x'_\pi\rangle) \\
\multicolumn{3}{l}{where\ N_i =\quad \lambda k'.let\ \langle z'_\theta, z'_\pi\rangle := \uparrow_c k'\ in\ let\ \langle x_1; \ldots; x_n\rangle_n := y_\phi\ in\ x_i\ (\downarrow_c \langle x'_\theta, z'_\pi\rangle)}
\end{array}$$

$$\begin{array}{rcl}
M'_i & \to_p^* & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in \\
& & \qquad let\ \langle y_\phi, y\rangle := \langle \pi_{11}(z), \pi_i^n(\pi_{21}(z))\rangle\ in\ y\ (\downarrow_c \langle\langle N'_1, \ldots, N'_n\rangle_n, z_\pi\rangle) \\
\multicolumn{3}{l}{where\ N'_i =\quad \lambda k'.let\ \langle z'_\theta, z'_\pi\rangle := \uparrow_c k'\ in\ let\ \langle x_1; \ldots; x_n\rangle_n := y_\phi\ in\ x_i\ (\downarrow_c \langle x_\theta, z'_\pi\rangle)}
\end{array}$$

$$\begin{array}{rcl}
M'_i & \to_p^* & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in\ \pi_i^n(\pi_{21}(z))\ (\downarrow_c \langle\langle N''_1, \ldots, N''_n\rangle_n, z_\pi\rangle) \\
\multicolumn{3}{l}{where\ N''_i = \quad \lambda k'.let\ \langle z'_\theta, z'_\pi\rangle := \uparrow_c k'\ in\ let\ \langle x_1; \ldots; x_n\rangle_n := \pi_{11}(z)\ in\ x_i\ (\downarrow_c \langle x_\theta, z'_\pi\rangle)}
\end{array}$$

Finally,

$$\begin{array}{rcl}
\dot{f} & \to_p^* & \lambda z k.let\ \langle x_\theta, x_\pi\rangle := \uparrow_c k\ in\ \pi_2(z)\ (\downarrow_c \langle\langle M''_1, \ldots, M''_n\rangle_n, x_\pi\rangle), where \\
M''_i & = & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in\ \pi_i^n(\pi_{21}(z))\ (\downarrow_c \langle\langle N''_1, \ldots, N''_n\rangle_n, z_\pi\rangle), and \\
N''_i & = & \lambda k'.let\ \langle z'_\theta, z'_\pi\rangle := \uparrow_c k'\ in\ let\ \langle x_1; \ldots; x_n\rangle_n := \pi_{11}(z)\ in\ x_i\ (\downarrow_c \langle x_\theta, z'_\pi\rangle)
\end{array}$$

On the other hand, let $g = h_\cong\ ;\ (Id_{D^n} \times \mathsf{case})\ ;\ \mathsf{case}$.

$$\begin{array}{rcll}
\lceil Id_{D^n} \times \mathsf{case}\rceil & \to_p^* & \lambda x.\langle \pi_1(x), M_{\mathsf{case}}[z := \pi_2(x)]\rangle & (B.7) \\
\lceil h_\cong; (Id_{D^n} \times \mathsf{case})\rceil & \to_p^* & \lambda z.\langle \pi_1(M_\cong), M_{\mathsf{case}}[z := \pi_2(M_\cong)]\rangle & (B.2) \\
& \to_p^* & \lambda z.\langle \pi_{11}(z), M_{\mathsf{case}}[z := \langle \pi_{21}(z), \pi_2(z)\rangle]\rangle &
\end{array}$$

$$\begin{array}{rcl}
Hence,\ \dot{g} \to_p^* & \lambda z.M_{\mathsf{case}}[z := \langle \pi_{11}(z), M_{\mathsf{case}}[z := \langle \pi_{21}(z), \pi_2(z)\rangle]\rangle\ ] & (B.2) \\
= & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi\rangle := \uparrow_c k\ in & \\
& \quad let\ \langle y_\phi, y\rangle := \langle \pi_{11}(z), M_{\mathsf{case}}[z := \langle \pi_{21}(z), \pi_2(z)\rangle]\rangle\ in & \\
& \qquad y\ (\downarrow_c \langle\langle M_1, \ldots, M_n\rangle_n, x_\pi\rangle) & \\
with\ M_i = & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in & \\
& \quad let\ \langle x_1; \ldots; x_n\rangle_n := y_\phi\ in\ x_i\ (\downarrow_c \langle x_\theta, z_\pi\rangle) &
\end{array}$$

$$\begin{array}{rcl}
\dot{g} \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi\rangle := \uparrow_c k\ in & \\
& \quad M_{\mathsf{case}}[z := \langle \pi_{21}(z), \pi_2(z)\rangle]\ (\downarrow_c \langle\langle N_1, \ldots, N_n\rangle_n, x_\pi\rangle) & \\
with\ N_i = & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in & \\
& \quad let\ \langle x_1; \ldots; x_n\rangle_n := \pi_{11}(z)\ in\ x_i\ (\downarrow_c \langle x_\theta, z_\pi\rangle) &
\end{array}$$

$$\begin{array}{rcl}
\dot{g} \to_p^* & \lambda z.\lambda k.let\ \langle x_\theta, x_\pi\rangle := \uparrow_c k\ in & \\
& \quad let\ \langle x'_\theta, x'_\pi\rangle := \langle\langle N_1, \ldots, N_n\rangle_n, x_\pi\rangle\ in & \\
& \quad let\ \langle y_\phi, y\rangle := \langle \pi_{21}(z), \pi_2(z)\rangle\ in\ y\ (\downarrow_c \langle\langle M'_1, \ldots, M'_n\rangle_n, x'_\pi\rangle) & \\
where\ M'_i = & \lambda k'.let\ \langle z_\theta, z_\pi\rangle := \uparrow_c k'\ in\ let\ \langle x_1; \ldots; x_n\rangle_n := y_\phi\ in & \\
& \quad x_i\ (\downarrow_c \langle x'_\theta, z_\pi\rangle)\ . &
\end{array}$$

Also,

$$\begin{array}{rl}
\dot{g}\to_p^* & \lambda z.\lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in \\
& \quad let\ \langle\!|y_\phi, y|\!\rangle := \langle\!|\pi_{21}(z), \pi_2(z)|\!\rangle\ in\ y\ \left(\downarrow_c \langle\!|\langle\!|M'_1, \ldots, M'_n|\!\rangle_n, x_\pi|\!\rangle\right) \\
\text{where } M'_i = & \lambda k'.let\ \langle\!|z_\theta, z_\pi|\!\rangle :=\uparrow_c k'\ in\ let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := y_\phi\ in \\
& \quad x_i\ \left(\downarrow_c \langle\!|\langle\!|N_1, \ldots, N_n|\!\rangle_n, z_\pi|\!\rangle\right)
\end{array}$$

$$\begin{array}{rl}
\dot{g}\to_p^* & \lambda z.\lambda k.let\ \langle\!|x_\theta, x_\pi|\!\rangle :=\uparrow_c k\ in\ \pi_2(z)\ \left(\downarrow_c \langle\!|\langle\!|M''_1, \ldots, M''_n|\!\rangle_n, x_\pi|\!\rangle\right) \\
\text{where } M''_i = & \lambda k'.let\ \langle\!|z_\theta, z_\pi|\!\rangle :=\uparrow_c k'\ in\ let\ \langle\!|x_1; \ldots; x_n|\!\rangle_n := \pi_{21}(z)\ in \\
& \quad x_i\ \left(\downarrow_c \langle\!|\langle\!|N_1, \ldots, N_n|\!\rangle_n, z_\pi|\!\rangle\right) \\
\to_p^* & \lambda k'.let\ \langle\!|z_\theta, z_\pi|\!\rangle :=\uparrow_c k'\ in\ \pi_i^n(\pi_{21}(z))\ \left(\downarrow_c \langle\!|\langle\!|N_1, \ldots, N_n|\!\rangle_n, z_\pi|\!\rangle\right)
\end{array}$$

Finally $\dot{g}$ reduces on the same term as $\dot{f}$, thus $f = g$ □

Appendix B

Lambda calculus with pairs

B.1 Lambda calculus with pairs

In this appendix we recall the lambda calculus with pairs (or λ_P-calculus), and we detail its different variants that we may use. We parametrise it with some $\mathcal{P} = (\mathcal{A}, \mathcal{A}_\mathcal{T}, \mathcal{T}, \mathcal{R}, \mathcal{E})$ where:

- $\mathcal{A}$ is a set of *atomic terms*,
- $\mathcal{A}_\mathcal{T}$ is a set of *atomic types*.

The set of terms $\mathcal{L}(\mathcal{P})$ and the set of types $\mathcal{L}_\mathcal{T}(\mathcal{P})$ are then given by the grammars of Fig. B.1. Reduction rules include the usual β-reduction and projections, as well as the atomic reduction of $\mathcal{R} \subseteq \mathcal{L}(\mathcal{P}) \times \mathcal{L}(\mathcal{P})$. There is also a set $\mathcal{T} \subseteq \mathcal{A} \times \mathcal{A}_\mathcal{T}$ of axiomatic typing rules, and an equivalence relation $\mathcal{E}$ on types that contextually closed. We may write $A = B$ for $(A, B) \in \mathcal{E}$. The whole calculus is described in Fig. B.1.

The reduction relation $\to_p$ in the λ_P-calculus is the contextual closure of $\twoheadrightarrow_p$. As usual, we write $\to_p^*$ the transitive closure of $\to_p$, and $\simeq_p$ its reflexive symmetric and transitive closure.

A *value* in λ_P-calculus is an abstraction or a pair:

$$V := \quad \lambda x.M \quad | \quad \langle\!\langle M, N \rangle\!\rangle$$

We write $\twoheadrightarrow_v$ the weak *call-by-value* strategy (*c.b.v.* for short) in the λ_P-calculus:

$$(\lambda x.M)\ V \ \twoheadrightarrow_v \ M[x := V] \qquad \pi_i(\langle\!\langle M_1, M_2 \rangle\!\rangle) \ \twoheadrightarrow_v \ M_i$$

$$\frac{M \twoheadrightarrow_v M'}{MN \twoheadrightarrow_v M'N} \qquad \frac{N \twoheadrightarrow_v N'}{(\lambda x.M)N \twoheadrightarrow_v (\lambda x.M)N'} \qquad \frac{(M, N) \in \mathcal{R}}{M \twoheadrightarrow_v N}$$

Well-designed parameter The simply typed lambda calculus with pairs satisfies the uniqueness of typing and the subject reduction property. We define a

Terms. $M, N, P := \dot{a} \mid x \mid \lambda x.M \mid MN \mid \langle\!\langle M, N \rangle\!\rangle \mid \pi_i(M)$ $(i \in \{1,2\})$ where $\dot{a} \in \mathcal{A}$

Types. $A, B := \dot{T} \mid A \to B \mid A \times B$ where $\dot{T} \in \mathcal{A}_{\mathcal{T}}$

Contexts. $\Gamma := \{x_1 : A_1, \ldots, x_k : A_k\}$

Reduction rules: $M \twoheadrightarrow_p N$ if $(M, N) \in \mathcal{R}$

$$(\lambda x.M)\ N \twoheadrightarrow_p M[x := N] \qquad \pi_i(\langle\!\langle M_1, M_2 \rangle\!\rangle) \twoheadrightarrow_p M_i$$
$$\text{If } x \notin \mathrm{fv}(M),\quad \lambda x.Mx \twoheadrightarrow_p M \qquad \langle\!\langle \pi_1(M), \pi_2(M) \rangle\!\rangle \twoheadrightarrow_p M$$

Typing rules:
$$\frac{x : A \in \Gamma}{\Gamma \vdash_p x : A}\mathbf{ax} \qquad \frac{(M, A) \in \mathcal{T}}{\Gamma \vdash_p M : A}\mathbf{ax}_{\mathcal{T}} \qquad \frac{\Gamma \vdash_p M : A \quad \Gamma \vdash_p A = B}{\Gamma \vdash_p M : B}\mathbf{subs}$$

$$\frac{\Gamma, x : A \vdash_p M : B}{\Gamma \vdash_p \lambda x.M : A \to B}{\to}\mathbf{intro} \qquad \frac{\Gamma \vdash_p M : A \quad \Gamma \vdash_p N : B}{\Gamma \vdash_p \langle\!\langle M, N \rangle\!\rangle : A \times B}{\times}\mathbf{intro}$$

$$\frac{\Gamma \vdash_p M : A \to B \quad \Gamma \vdash_p N : A}{\Gamma \vdash_p MN : B}{\to}\mathbf{elim} \qquad \frac{\Gamma \vdash_p M : A_1 \times A_2}{\Gamma \vdash_p \pi_i(M) : A_i}{\times}\mathbf{elim}$$

Figure B.1: Lambda calculus with pairs parametrised with $\mathcal{P}$

notion of good design for $\mathcal{P}$ that preserves those properties for the parametrised λ_P-calculus.

Definition B.1.1

The parameter $\mathcal{P} = (\mathcal{A}, \mathcal{A}_{\mathcal{T}}, \mathcal{T}, \mathcal{R}, \mathcal{E})$ is *well designed* when:

- if $(M, N) \in \mathcal{R}$, then $\Gamma \vdash_p M : B$ implies $\Gamma \vdash_p N : B$ for each context Γ and each type B,
- $\mathcal{T}$ associates only one type to each atomic term: if $(\dot{a}, \dot{T}) \in \mathcal{T}$ and $(\dot{a}, \dot{T}') \in \mathcal{T}$, then $\dot{T} = \dot{T}'$.
- $\mathcal{E}$ is stable by sub-type: $A_1 \times A_2 = B_1 \times B_2$ (or $A_1 \to A_2 = B_1 \to B_2$) implies $A_i = B_i$ (for $i = 1$ and $i = 2$).

Remark that the minimal lambda-calculus with pairs (where $\mathcal{A}, \mathcal{R}, \mathcal{T}$ and $\mathcal{E}$ are empty, and where there is only one atomic type) is well-designed.

Lemma B.1.1 (Substitution)**.** *For any terms M, N, any types A, B and any context Γ, if $\Gamma, x : A \vdash_p M : B$ and $\Gamma \vdash_p N : A$ then $\Gamma \vdash_p M[x := N] : B$.*

Proof : *By induction on (the derivation of) $\Gamma, x : A \vdash_p M : B$.* □

Lemma B.1.2 (Weakening)**.** *For any terms M such that $x \notin \mathrm{fv}(M)$, if $\Gamma, x : A \vdash_p M : B$ then $\Gamma \vdash_p M : B$.*

PROOF : *By induction on* $\Gamma, x : A \vdash_p M : B$. □

Lemma B.1.3 (Subject Reduction)**.** *If* $\mathcal{P}$ *is well-designed, then for any terms* M, N*, any context* Γ *and any type* B*,*

$$\begin{cases} \Gamma \vdash_p M : B \\ M \to_p M' \end{cases} \implies \Gamma \vdash_p M' : B$$

PROOF : *By induction on* $\Gamma, x : A \vdash_p M : B$*. If the last step of the derivation uses the rule* SUBS*, or if the redex involved in* $M \to_p M'$ *is a strict sub-term of* M *then we conclude with the induction hypothesis. Otherwise,* $M \twoheadrightarrow_p M'$ *and the structure of* M *determines the last rule involved in the derivation of* $\Gamma \vdash_p M : B$*:*

- *If* $(M, M') \in \mathcal{R}$ *then* $\Gamma \vdash_p M' : B$ *since* $\mathcal{P}$ *is well-designed.*
- *If* $M = (\lambda x.M_0)N$ *and* $M' = M_0[x := N]$*, with* $\dfrac{\Gamma \vdash_p \lambda x.M_0 : A \to B \quad \Gamma \vdash_p N : A}{\Gamma \vdash_p (\lambda x.M_0)N : B}$*, then* $\Gamma \vdash_p M_0[x := N] : B$ *by Lem. B.1.1.*
- *If* $M = \lambda x.M'x$*, with* $x \notin \mathrm{fv}(M')$*, and*

$$\frac{\Gamma, x : B_1 \vdash_p Mx : B_2}{\Gamma \vdash_p \lambda x.Mx : B_1 \to B_2} \qquad \text{where } B_1 \to B_2 = B,$$

  then the derivation of $\Gamma, x : B_1 \vdash_p Mx : B_2$ *necessarily ends with*

$$\dfrac{\dfrac{\Gamma, x : B_1 \vdash_p M : A_1 \to A_2 \qquad \Gamma, x : B_1 \vdash_p x : A_1}{\Gamma, x : B_1 \vdash_p Mx : A_2}\ {\to}\mathbf{elim}}{\Gamma, x : B_1 \vdash_p Mx : B_2}\ \mathbf{subs}$$

  The dashed line represent zero, one or several deduction step with the rule SUBS*, also* $A_2 = B_2$*. Moreover* $\Gamma, x : B_1 \vdash_p x : A_1$ *implies* $A_1 = B_1$*, hence* $B = B_1 \to B_2 = A_1 \to A_2$ *. Also* $\Gamma, x : B_1 \vdash_p M : A_1 \to A_2$ *implies* $\Gamma, x : B_1 \vdash_p M : B$ *(Lem. B.1.2).*

- *If* $M = \pi_i(\langle\!| N_1, N_2 |\!\rangle)$*, with* $M' = N_i$*, and*

$$\frac{\Gamma \vdash_p \langle\!| N_1, N_2 |\!\rangle : B_1 \times B_2}{\Gamma \vdash_p \pi_i(\langle\!| N_1, N_2 |\!\rangle) : B_i} \qquad \text{where } B_i = B,$$

  then the derivation of $\Gamma \vdash_p \langle\!| N_1, N_2 |\!\rangle : B_1 \times B_2$ *necessarily ends with*

$$\dfrac{\dfrac{\Gamma \vdash_p N_1 : A_1 \qquad \Gamma \vdash_p N_2 : A_2}{\Gamma \vdash_p \langle\!| N_1, N_2 |\!\rangle : A_1 \times A_2}\ {\times}\mathbf{intro}}{\Gamma \vdash_p \langle\!| N_1, N_2 |\!\rangle : B_1 \times B_2}\ \mathbf{subs}$$

  Also $B_1 \times B_2 = A_1 \times A_2$*, which means that* $B_i = A_i$ *if* $\mathcal{P}$ *is well-designed. Hence we can derive* $\Gamma \vdash_p N_i : B_i$*.*

- *If* $M = \langle\!| \pi_1(M'), \pi_2(M') |\!\rangle$*, and*

$$\frac{\Gamma \vdash_p \pi_1(M') : B_1 \quad \Gamma \vdash_p \pi_2(M') : B_2}{\Gamma \vdash_p \langle\!| \pi_1(M'), \pi_2(M') |\!\rangle : B_1 \times B_2} \qquad \text{where } B_1 \times B_2 = B,$$

  then the derivation of $\Gamma \vdash_p \pi_i(M') : B_i$ *necessarily ends with*

$$\dfrac{\dfrac{\Gamma \vdash_p M' : A_1 \times A_2}{\Gamma \vdash_p \pi_i(M') A_i}\ \times\textbf{elim}}{\Gamma \vdash_p \pi_i(M') B_i}\ \textbf{subs}$$

Also $A_i = B_i$ *and thus* $B = B_1 \times B_2 = A_1 \times A_2$. *Hence* $\Gamma \vdash_p M' : B$. □

Lemma B.1.4 (Unique type)**.** *If* $\mathcal{P}$ *is well designed, then* $\Gamma \vdash_p M : B$ *and* $\Gamma \vdash_p M : B'$ *imply* $B = B'$

PROOF : *By induction on* $\Gamma \vdash_p M : B$. □

Generalised pairs In the following we may need tuples of terms a numerable products of types. Therefore we extend the notation of pairs and product:

$$\text{For any } k \geq 1, \qquad \begin{array}{l} A_1 \times \cdots \times A_k = ((A_1 \times A_2) \times \cdots) \times A_k \\ \langle\!\langle x_1; \ldots; x_k \rangle\!\rangle_k = \langle\!\langle \langle\!\langle x_1, x_2 \rangle\!\rangle, \cdots \rangle\!\rangle, x_k \rangle\!\rangle \end{array}$$

If all A_i's are the same type A we may write A^n the n-ary product $A \times \cdots \times A$. We also generalise the projection:

$$\begin{cases} \pi_k^k(M) = \pi_2(M) & \text{if } k > 1 \\ \pi_1^1(M) = M & \\ \pi_i^k(M) = \pi_i^{k-1}(\pi_1(M)) & \text{if } 1 \leq i < k \end{cases}$$

Within these notations, the following rules (that generalise the rules $\times$**intro** and $\times$**elim**) are derivable:

$$\frac{\Gamma \vdash_p M_1 : A_1 \quad \cdots \quad \Gamma \vdash_p M_k : A_k}{\Gamma \vdash_p \langle\!\langle M_1, \ldots, M_k \rangle\!\rangle_k : A_1 \times \cdots \times A_k} \qquad \frac{\Gamma \vdash_p M : A_1 \times \cdots \times A_k}{\Gamma \vdash_p \pi_i^k(M) : A_i}$$

Moreover,

$$\pi_i^k(\langle\!\langle M_1, \cdots, M_k \rangle\!\rangle_n) \to_p^* M_i, \qquad \text{and} \qquad \langle\!\langle \pi_1^k(M), \cdots, \pi_k^k(M) \rangle\!\rangle_n \to_p^* M\ .$$

B.2 Lambda calculus generated by a CCC.

Cartesian closed categories exactly correspond to simply typed lambda calculus with pairs [AC03, Sec. 4.4]. Here we formalise the fact that we can use the language of lambda calculus with pairs to speak about the morphisms and the objects of a CCC.

In this section, $\mathbb{C}$ denotes a Cartesian closed category. In order to mimic the type notation for the n-ary product, we use the following notation for the objects of $\mathbb{C}$:

$$A_1 \times \cdots \times A_k = ((A_1 \times A_2) \times \cdots) \times A_k$$

It is the limit of the discrete diagram $A_1 \cdots A_k$ with the projection morphisms π_i^k, where

$$\pi_k^1 = \underbrace{\pi_1; \cdots; \pi_1}_{k-1 \text{ times}} \qquad ; \qquad \pi_k^i = \underbrace{\pi_1; \cdots; \pi_1}_{k-i \text{ times}}; \pi_2$$

If $k = 1$, the projection morphism is Id_{A_1}, and if $k = 0$ the k-ary product is $\mathbf{1}$, the terminal object.

B.2.1 Definition

The lambda-calculus generated by $\mathbb{C}$ consists in adding one atomic term $\dot{f}$ for each morphism f of $\mathbb{C}$and one atomic type $\dot{A}$ for each object A. We may write $\lceil \dot{f} \rceil$ for $\dot{f}$ when the expression of f is too long. It is the λ_P-calculus where:

- $\mathcal{A} = \{\dot{f} / f \text{ is a morphism of } \mathbb{C}\}$.
- $\mathcal{A}_{\mathcal{T}} = \{\dot{A} / A \text{ is an object of } \mathbb{C}\}$.
- $\mathcal{R} = \{(\dot{f}, \dot{g}) / f = g\} \cup \{ \quad (\dot{Id}_A\,,\, \lambda x.x) \quad / \quad Id_A : A \to A \quad \}$
  $\cup \{ \quad (\dot{\mathtt{ev}}\,,\, \lambda x.\pi_1(x)\pi_2(x)) \quad / \quad \mathtt{ev} : A \times B^A \to B \quad \}$
  $\cup \{ \quad (\dot{\Lambda}(\dot{f})\,,\, \lambda xy.\dot{f}\langle\!\langle x, y \rangle\!\rangle) \quad / \quad f : A \times C \to B \quad \}$
  $\cup \{ \quad (\lceil \dot{f}; \dot{g} \rceil\,,\, \lambda x.\dot{g}\ (\dot{f}x)) \quad / \quad f : A \to B,\ g : B \to C \quad \}$
  $\cup \{ \quad (\dot{\pi}_i\,,\, \lambda x.\pi_i(x)) \quad / \, i \in \{1,2\}\,,\, \pi_i : A_1 \times A_2 \to A_i \}$
  $\cup \{ (\lceil \dot{\langle} f, g \dot{\rangle} \rceil\,,\, \lambda x.\dot{\langle\!\langle} \dot{f}x, \dot{g}x \dot{\rangle\!\rangle}) \quad / \quad f : C \to A,\ g : C \to B \quad \}$
- $\mathcal{T} = \{(\dot{f}, \dot{B^A}) \ / \ f : A \to B\}$.
- $\mathcal{E}$ is the symmetric reflexive transitive and contextual closure of

$$\{(\dot{B^A}, \dot{A} \to \dot{B}) / A, B \text{ objects of } \mathbb{C}\} \cup \{(\lceil A \dot{\times} B \rceil, \dot{A} \times \dot{B}) / A, B \text{ objects of } \mathbb{C}\}\ .$$

This calculus is called the $\lambda_{\mathbb{C}}$-calculus. It enjoys the subject reduction property (Lem. B.1.3) and uniqueness of type (for typable terms, Lem: B.1.3) since $\mathcal{P}$ is well-designed.

Lemma B.2.1. *In the definition of the $\lambda_{\mathbb{C}}$-calculus, $\mathcal{P}$ is well designed.*

Proof : *We check that for any $(M, N) \in \mathcal{R}$, if $\Gamma \vdash_p M : B$ then $\Gamma \vdash_p N : B$ (the other points of Def. B.1.1 are trivially true):*

- *If $M = \dot{f}$ and $g = \dot{g}$ with $f = g : B \to C$ then necessarily $A = \dot{C^B}$ and thus $\Gamma \vdash_p \dot{g} : A$.*
- *If $M = \dot{\mathtt{ev}}$ and $N = \lambda x.\pi_1(x)\pi_2(x)$ with $\mathtt{ev} : A \times B^A \to B$, then necessarily $B = \dot{B^{A \times B^A}} = (\dot{A} \times (\dot{A} \to \dot{B})) \to \dot{A}$. Moreover we can derive*

$$\Gamma \vdash_p \lambda x.\pi_1(x)\pi_2(x) : (\dot{A} \times (\dot{A} \to \dot{B})) \to \dot{A}\ .$$

- *If $M = \dot{\Lambda}(\dot{f})$ and $N = \lambda xy.\dot{f}\langle\!\langle x, y \rangle\!\rangle$, with $f : C \times A \to B$, then $\Lambda(f)$ is a morphism of $C \to B^A$ and so $B = \dot{(B^A)^C} = \dot{C} \to \dot{A} \to \dot{B}$. Moreover we can derive*

$$\Gamma \vdash_p \lambda xy.\dot{f}\langle\!\langle x, y \rangle\!\rangle : \dot{C} \to \dot{A} \to \dot{B}\ .$$

- *If $M = \lceil \dot{f}; \dot{g} \rceil$ and $\lambda x.\dot{g}\ (\dot{f}x)$, with $f : A \to B$ and $g : B \to C$ then necessarily $B = \dot{A} \to \dot{C}$ and we can derive $\Gamma \vdash_p \lambda x.\dot{g}\ (\dot{f}x) : \dot{A} \to \dot{C}$.*
- *If $M = \dot{\pi}_i$ and $N = \lambda x.\pi_i(x)$ with $\pi_i : A_1 \times A_2 \to A_i$ then $\Gamma \vdash_p N : \dot{A}_1 \times \dot{A}_2 \to \dot{A}_i$.*
- *If $M = \lceil \dot{\langle} f, g \dot{\rangle} \rceil$ and $N = \lambda x.\dot{\langle\!\langle} \dot{f}x, \dot{g}x \dot{\rangle\!\rangle}$ with $f : C \to A$ and $g : C \to B$, then necessarily $B = \lceil \dot{(A \times B)^C} \rceil$. Hence we can derive $\Gamma \vdash_p \lambda x.\dot{\langle\!\langle} \dot{f}x, \dot{g}x \dot{\rangle\!\rangle} : B$* □

To any object and any morphism of $\mathbb{C}$ correspond respectively a type and a term of the $\lambda_{\mathbb{C}}$-calculus. We will see that the converse holds.

Lemma B.2.2. *For any type A of the $\lambda_{\mathbb{C}}$-calculus, there is a unique object A' of $\mathbb{C}$ such that $A = \dot{A}'$.*

Proof : *By structural induction on A.* □

In the following we may make the confusion between the object and the type, and write the object A' with A too:

$$A \text{ denotes the object } A \text{ and the type } \dot{A}.$$

In particular, $\mathbf{1}$ is the atomic type representing the terminal object of $\mathbb{C}$.

B.2.2 From terms to morphisms

Now we attribute a morphism to each typable term. For any context $\Gamma = \{x_1 : A_1, \dots, x_k : A_k\}$, we also write Γ the object $A_1 \times \dots \times A_k$ (in particular if Γ is the empty context, then Γ denotes the terminal object $\mathbf{1}$). Given a judgement $\Gamma \vdash_p M : B$, the morphism $\lfloor M \rfloor_\Gamma : \Gamma \to B$ is inductively defined in Fig. B.2. Notice that Lem. B.1.4 ensures that we do not need to specify B in the notation $\lfloor M \rfloor_\Gamma$. We might also denote this morphism by f_{M} when there is no ambiguity concerning the context Γ.

Admissible rules Remember that if to morphisms f and g are equal, then $\dot{f} \to_p \dot{g}$, by definition of $\mathcal{R}$. We extend the notion of reduction in order to get the converse.

Definition B.2.1 (*Admissible rule*)

Given two λ_P-terms N and N', we say that a reduction rule from M to N is *admissible* (what we write $N \to_{\mathbb{C}} N'$) when for any context Γ and any type B,

$$\Gamma \vdash_p N : B \qquad \Longrightarrow \qquad \begin{cases} \Gamma \vdash_p N' : B, \text{ and} \\ f_{\mathrm{N}} = f_{\mathrm{N}'} \end{cases}$$

Remark this notion of admissible reduction is contextually closed: a trivial induction on M ensures that

$$N \to_{\mathbb{C}} N' \quad \Longrightarrow \quad M[x := N] \to_{\mathbb{C}} M[x := N'] \ .$$

Also the admissible rules constitute a rewriting system. The following lemma shows that they include all reduction rules of the λ_P-calculus.

Lemma B.2.3. *For any terms M, N such that $M \to_p N$, and any context Γ and any type B such that $\Gamma \vdash_p M : B$,*

$$\Gamma \vdash_p N : B \qquad \textit{and} \qquad f_{\mathrm{M}} = f_{\mathrm{N}} \ .$$

Γ is the context $\{x_1 : A_1, \ldots, x_k : A_k\}$, and denotes also the object $A_1 \times \ldots \times A_k$.

$$\Gamma \vdash_p M : B \qquad \rightsquigarrow \qquad f_{\mathrm{M}} = \lfloor M \rfloor_\Gamma : \quad \Gamma \to B$$

- $\Gamma \vdash_p x_i : A_i (\mathbf{ax})$. Then $f_{x_i} = \pi_i^k$
- $\dfrac{f : A \to B \text{ in } \mathbb{C}}{\Gamma \vdash_p \dot{f} : A \to B}\mathbf{ax}_{\mathcal{T}}$. Then $\lfloor \dot{f} \rfloor_\Gamma = (!_\Gamma \,;\, \Lambda(\pi_2; f)) : \Gamma \to B^A$ (where $(\pi_2; f) : \mathbf{1} \times A \to B$).
- $\dfrac{\Gamma \vdash_p M : A \quad \Gamma \vdash_p A = B}{\Gamma \vdash_p M : B}\mathbf{subs}$. By induction, $f_{\mathrm{M}} : \Gamma \to A$.
  By uniqueness (Lem. B.2.2), types A and B represent the same object in $\mathbb{C}$, so we use the same morphism f_{M} for $\Gamma \vdash_p M : B$.
- $\dfrac{\Gamma, x_{k+1} : A_{k+1} \vdash_p M : B}{\Gamma \vdash_p \lambda x.M : A_{k+1} \to B}{\to}\mathbf{intro}$. By induction, $f_{\mathrm{M}} : \Gamma \times A_{k+1} \to B$.
  Then $f_{\lambda x_{k+1}.M} = \Lambda(f_{\mathrm{M}}) : \Gamma \to B^{A_{k+1}}$.
- $\dfrac{\Gamma \vdash_p M_1 : B_1 \quad \Gamma \vdash_p M_2 : B_2}{\Gamma \vdash_p \langle\!| M_1, M_2 |\!\rangle : B_1 \times B_2}{\times}\mathbf{intro}$. By induction, $f_{\mathrm{M}_i} : \Gamma \to B_i$ (for both i).
  Then $f_{\langle\!| M_1, M_2 |\!\rangle} = \langle f_{\mathrm{M}_2}, f_{\mathrm{M}_2} \rangle : \Gamma \to B_1 \times B_2$.
- $\dfrac{\Gamma \vdash_p M : A \to B \quad \Gamma \vdash_p N : A}{\Gamma \vdash_p MN : B}{\to}\mathbf{elim}$. By induction, $f_{\mathrm{M}} : \Gamma \to B^A$ and $f_{\mathrm{N}} : \Gamma \to A$.
  Then $f_{\mathrm{M}n} = \langle f_{\mathrm{M}}, f_{\mathrm{N}} \rangle; \mathtt{ev} : \Gamma \to B$.
- $\dfrac{\Gamma \vdash_p M : B_1 \times B_2}{\Gamma \vdash_p \pi_i(M) : B_i}{\times}\mathbf{elim}$. By induction, $f_{\mathrm{M}} : \Gamma \to B_1 \times B_2$.
  Then $f_{\pi_i(M)} = f_{\mathrm{M}}; \pi_i : \Gamma \to B_i$.

Figure B.2: From typable terms to morphisms

PROOF : *First,* $\Gamma \vdash_p N : B$ *by subject reduction (Lem. B.1.3). Then the equality* $f_{\mathrm{M}} = f_{\mathrm{N}}$ *is proved by induction on* M*.*

Remark B.2.1 . In terms of binary relations, the previous lemma means that

$$\rightarrow_p \subseteq \rightarrow_{\mathbb{C}} \ .$$

Moreover, the transitivity of the equality on morphisms ensures that $\rightarrow_{\mathbb{C}}^* \subseteq \rightarrow_{\mathbb{C}}$. Hence $\rightarrow_p^* \subseteq \rightarrow_{\mathbb{C}}$. Yet, remark that $\simeq_p \not\subseteq \rightarrow_{\mathbb{C}}$, since $\leftarrow_p \not\subseteq \rightarrow_{\mathbb{C}}$. For instance, if $\vdash_p M_i : A_i$ (for $i = 1$ and $i = 2$), then

$$\vdash_p \langle\!| M_1, M_2 |\!\rangle : A_1 \times A_2 \ , \quad \text{and} \quad \lambda x.\langle\!| M_1, M_2 |\!\rangle x \ \rightarrow_p \ \langle\!| M_1, M_2 |\!\rangle \ .$$

However, $\langle\!| M_1, M_2 |\!\rangle \not\rightarrow_{\mathbb{C}} \lambda x.\langle\!| M_1, M_2 |\!\rangle x$, since $\lambda x.\langle\!| M_1, M_2 |\!\rangle x$ is not typable.

The following proposition is quite obvious, but it is a key point in the use of the syntax of the lambda calculus with pairs to show equalities of morphisms in a CCC.

Proposition B.2.4. *For any objects* A*,* B *of* $\mathbb{C}$*, and any morphisms* f *and* g *in* $\mathbb{C}[A, B]$*,*

$$\exists M \in \mathcal{L}(\mathcal{P}) \ , \quad (\dot{f} \rightarrow_{\mathbb{C}} M \ \wedge \ \dot{g} \rightarrow_{\mathbb{C}} M) \qquad \Longrightarrow \qquad f = g. \tag{B.1}$$

PROOF : *First,* $\dot{B^A} = \dot{A} \rightarrow \dot{B}$ $(= A \rightarrow B)$*, also* $\vdash_p \dot{f} : A \rightarrow B$ *and* $\vdash_p \dot{g} : A \rightarrow B$*. Then* $\vdash_p M : A \rightarrow B$ *and* $\lfloor \dot{f} \rfloor = \lfloor M \rfloor = \lfloor \dot{g} \rfloor$*. By definition, it means that* $\Lambda(\pi_2; f) = \Lambda(\pi_2; g)$ *. Hence* $\pi_2; f = \pi_2; g : \mathbf{1} \times A \rightarrow B$*. Since* π_2 *is an isomorphism between* $\mathbf{1} \times A$ *and* A *(with the inverse* $\langle !A, Id_A \rangle$*), this implies that* $f = g$*.* □

Finally, this function from typable terms to morphisms of $\mathbb{C}$ is to, some extent, reversible with the interpretation of morphisms by atomic terms.

Proposition B.2.5. *Let* $\Gamma = \{x_1 : A_1, \ldots, x_k : A_k\}$*. Remember that* $\Gamma \vdash_p M : B$ *implies* $f_{\mathrm{M}} = \lfloor M \rfloor_\Gamma : \Gamma \rightarrow B$ *. Moreover,*

$$\dot{f_{\mathrm{M}}} \quad \rightarrow_p^* \quad \lambda z.M[x_i := \pi_i^k(z)]_{i=1}^k.$$

The proof relies on the following lemmas, and then proceeds by induction on the derivation of $\Gamma \vdash_p M : B$.

Lemma B.2.6 (Substitution is composition)**.** *Given two morphisms* $f : C \rightarrow A$ *and* $g : A \rightarrow B$*, and two terms* M, N*, then*

$$\left\{ \begin{array}{l} \dot{f} \rightarrow_p^* \lambda x.N \\ \dot{g} \rightarrow_p^* \lambda z.M \end{array} \right. \qquad \Longrightarrow \qquad \left\{ \begin{array}{l} x : C \vdash_p N : A \\ z : A \vdash_p M : B \\ \lceil \dot{f}; \dot{g} \rceil \rightarrow_p^* \lambda x.M[z := N] \end{array} \right. \tag{B.2}$$

Lemma B.2.7 (Application is evaluation)**.** *Given two morphisms $f : C \to B^A$ and $g : C \to A$, and two terms M and N, then*

$$\begin{cases} \dot{f} \to_p^* \lambda x.M \\ \dot{g} \to_p^* \lambda x.N \end{cases} \quad \Longrightarrow \quad \begin{cases} x : C \vdash_p M : A \to B \\ x : C \vdash_p N : A \\ \ulcorner \dot{\langle} f, g \dot{\rangle} ; \mathtt{ev} \urcorner \to_p^* \lambda x.MN \end{cases} \tag{B.3}$$

The following lemma requires to prove with a trivial induction that

$$z : A \times C \vdash_p M[x := \pi_1(z)][y := \pi_2(z)] : B \implies x : C, y : A \vdash_p M : B$$

Lemma B.2.8 (Abstraction is curried form)**.** *Given a morphism $f : C \times A \to B$ and a term M,*

$$\dot{f} \to_p^* \lambda z.M[x := \pi_1(z)][y := \pi_2(z)] \quad \Longrightarrow \quad \begin{cases} x : C, y : A \vdash_p M : B \\ \Lambda(\dot{f}) \to_p^* \lambda x.\lambda y.M \end{cases} \tag{B.4}$$

Lemma B.2.9 (Pair is pairing)**.** *Given two morphisms $f_i : C \to A_i$ (for $i = 1, 2$), then for any M_1, M_2,*

$$\dot{f}_i \to_p^* \lambda x.M_i \ \text{ for both } i \quad \Longrightarrow \quad \begin{cases} x : C \vdash_p M_i : B_i \text{ for both } i \\ \ulcorner \dot{\langle} f, g \dot{\rangle} \urcorner \to_p^* \lambda x.\langle\!| M_1, M_2 |\!\rangle \end{cases} \tag{B.5}$$

Lemma B.2.10 (Projection is projection)**.** *Given a morphism $f : C \to B_1 \times B_2$, then for any term M, and any $i \in \{1, 2\}$,*

$$\dot{f} \to_p^* \lambda x.M \quad \Longrightarrow \quad \begin{cases} x : C \vdash_p M : B_1 \times B_2 \\ \ulcorner \dot{f}; \pi_i \urcorner \to_p^* \lambda x.\pi_i(M) \end{cases} \tag{B.6}$$

Corollary B.2.11. *Given two morphism f_1, f_2 from A_i to B_i (for $i = 1, 2$), then for any terms M_1, M_2,*

$$\begin{cases} \dot{f}_1 \to_p^* \lambda x.M_1 \\ \dot{f}_2 \to_p^* \lambda x.M_2 \end{cases} \quad \Longrightarrow \quad \begin{cases} x : A_i \vdash_p M_i : B_i \\ \ulcorner f_1 \dot{\times} f_2 \urcorner \to_p^* \lambda z.\langle\!| M_1[x := \pi_1(z)], M_2[x := \pi_2(z)] |\!\rangle \end{cases} \tag{B.7}$$

Index

Bibliography

[AC03] Roberto Amadio and Pierre-Louis Curien. *Domains and Lambda-calculi*. Cambridge University Press, 2003.

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures*. M.I.T. Press, 1991.

[AMR06] Ariel Arbiser, Alexandre Miquel, and Alejandro Ríos. A lambda-calculus with constructors. In *RTA*, pages 181–196, 2006.

[AMR09] Ariel Arbiser, Alexandre Miquel, and Alejandro Ríos. The lambda-calculus with constructors: Syntax, confluence and separation. *Journal of Functional Programming*, 19(5):581–631, 2009.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1984.

[BB64] Edmund C. Berkeley and Daniel G. Bobrow. *The Programming Language LISP: Its Operation and Applications*. MIT Press, 1964.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[BDCPR79] Corrado Böhm, Mariangiola Dezani-Ciancaglini, P. Peretti, and Simona Ronchi Della Rocca. A discrimination algorithm inside lambda-beta-calculus. *Theor. Comput. Sci.*, 8:265–292, 1979.

[Cho57] Noam Chomsky. *Syntactic Structures*. The Hague/Paris: Mouton, 1957.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[Chu36] Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Annals of Mathematics Studies (AM-6). Princeton University Press, 1941.

[Cir01] Horatiu Cirstea. *Calcul de Réécriture : Fondements et Applications*. PhD thesis, Université de Nancy, 2001.

[CK98] Horatiu Cirstea and Claude Kirchner. The rewriting calculus as a semantics of elan. In *ASIAN*, pages 84–85, 1998.

[CK00] Horatiu Cirstea and Claude Kirchner. The simply typed rewriting calculus. *Electr. Notes Theor. Comput. Sci.*, 36, 2000.

[dB91] Nicolaas Govert de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, 1991.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3-4):259–288, 1993.

[FM09] Germain Faure and Alexandre Miquel. A categorical semantics for the parallel lambda-calculus. Technical Report 7063, INRIA, October 2009.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Programming Language Design and Implementation*, pages 268–277, 1991.

[Gal98] Jean H. Gallier. Typing untyped lambda-terms, or reducibility strikes again! *Annals of Pure and Applied Logic*, 91(2-3):231–270, 1998.

[Ghi96] Silvia Ghilezan. Strong normalization and typability with intersection types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir01] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. In *CSL*, page 38, 2001.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[God33] Kurt Godel. Zum entscheidungsproblem des logischen funktionenkalküls. *Monatshefte für Mathematik*, 40(1):433–443, 1933.

[GS90] Carl A. Gunter and Dana S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. 1990.

[Gö34] Kurt Gödel. *On Undecidable Propositions of Formal Mathematical Systems*, pages 39–74. B. Meltzer, 1934. Lecture Notes Taken by Kleene and Rosser at the Institute for Advanced Study.

[HS97] Martin Hofmann and Thomas Streicher. Continuation models are universal for lambda-mu-calculus. In *LICS*, pages 387–395, 1997.

[Jay04] C. Barry Jay. The pattern calculus. *ACM TOPLAS*, 26(6):911–937, 2004.

[Jay09] Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.

[JK06] C. Barry Jay and Delia Kesner. Pure pattern calculus. In *ESOP*, pages 100–114, 2006.

[Kle36a] Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.

[Kle36b] Stephen C. Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

[Kri93] Jean-Louis Krivine. *Lambda-Calculus, Types and Models*. Ellis Horwood Ltd, 1993.

[Laf91] Yves Lafont. Negation versus implication. In *Logical Frameworks*, page 223, 1991.

[Lan71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[LRS93] Yves Lafont, Bernhard Reus, and Thomas Streicher. Continuation semantics or expressing implication by negation. Technical report, University of Munich, 1993.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[Mit86] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions (summary). In *LISP and Functional Programming*, pages 308–319, 1986.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, 1994.

[Par92] Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, pages 190–201, 1992.

[Par93] Michel Parigot. Strong normalization for second order classical natural deduction. In *LICS*, pages 39–46, 1993.

[Pet11] Barbara Petit. Semantics of typed lambda-calculus with constructors. *Logical Methods in Computer Science*, 7(1:2), 2011.

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical report, Carnegie Mellon University, 1991.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[Plo77] Gordon D. Plotkin. Lcf considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[Rib07a] Colin Riba. *Définitions par réécriture dans le lambda-calcul : confluence, réductibilité et typage*. PhD thesis, Université de Nancy, 2007.

[Rib07b] Colin Riba. On the stability by union of reducibility candidates. In *FoSSaCS*, pages 317–331, 2007.

[RS98] Bernhard Reus and Thomas Streicher. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[Sco70] Dana Scott. Outline of a mathematical theory of computation. Technical report, Princeton University, 1970.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992.

[SHLG94] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edwrad R. Griffor. *Mathematical Theory of Domains*. Number 22 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, New York, NY, USA, 1994.

[Tai67] William W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[TC87] Val Tannen and Thierry Coquand. Extensional models for polymorphism. In *TAPSOFT, Vol.2*, pages 291–307, 1987.

[Tes53] Lucien Tesnière. *Esquisse d'une syntaxe structurale*. Klincksieck, 1953.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *London Mathematical Society*, volume 42(2), pages 230–265, 1936.

[Tur37] Alan M. Turing. Computability and lambda-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937.

[Urz03] Pawel Urzyczyn. A simple proof of the undecidability of strong normalisation. *Mathematical Structures in Computer Science*, 13(1):5–13, 2003.

[vB92] Steffen van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[vO90] Vincent van Oostrom. Lambda calculus with patterns. Technical report, Vrije Universiteit, Amsterdam, 1990.

[Wel94] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. pages 176–185, 1994.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[Win93] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

Abstract

The lambda calculus with constructors (or $\lambda_{\mathscr{C}}$-calculus) was introduced by Arbiser, Miquel and Ríos in the early 2000's as an extension of lambda calculus with pattern matching features. It decomposes the pattern matching "à la ML" into a case-analysis on constant constructors (in the spirit of the `case` instruction in Pascal), and a commutation rule between case construction and application. This commutation rule between two different kinds of constructions designs a surprising computational behaviour, *a priori* not compatible with usual typing intuitions. However the whole calculus was proved confluent, and as far as we know it is the only calculus with pattern matching facilities that enjoys the *separation property* (a version of Böhm's lemma).

In this thesis, we first present the $\lambda_{\mathscr{C}}$-calculus, and we propose a polymorphic type system for it. Then we develop a realisability model, based on Girard's reducibility candidates. This leads to a strong normalisation result for the typed calculus. It also guaranties that the type system prevents match failure.

Next we focus on semantics for the untyped $\lambda_{\mathscr{C}}$-calculus. We start with defining a generic notion of $\lambda_{\mathscr{C}}$-models in Cartesian closed categories, that we show to be sound. We then establish the syntactic model in the category of partial equivalence relations, and deduce a completeness result from it.

Finally, we consider a translation of the lambda calculus with constructors into the pure lambda lambda calculus (or well-known variants of it) relying on *continuation passing style* techniques. We introduce it throw a stack abstract machine for $\lambda_{\mathscr{C}}$-calculus. This abstract machine (Sec. 5.1) can also be taken as a first presentation of the calculus itself, as it might give some intuitions about how it computes. After showing the simulation of $\lambda_{\mathscr{C}}$-calculus by pure lambda calculus, we come back to categorical models: we use the CPS-translation to obtain a $\lambda_{\mathscr{C}}$-model from any continuation model of lambda calculus. There comes out an equation in categories (in the spirit of lambda calculus characteristic equation $D \cong D^D$), whose every solution provides a $\lambda_{\mathscr{C}}$-model. Resolving this equation in domains category (using Scott's construction of D_∞ domain) finally leads to a non syntactic model of untyped lambda calculus with constructors.