Logical Methods in Computer Science
Vol. 7 (1:2) 2010, pp. 1–24
www.lmcs-online.org

Submitted Nov. 22, 2009
Published Mar. 16, 2011

SEMANTICS OF TYPED LAMBDA-CALCULUS WITH CONSTRUCTORS

BARBARA PETIT

LIP - ENS Lyon, 46 Allée d'Italie, 69364 Lyon, France
URL: `http://perso.ens-lyon.fr/barbara.petit`

Abstract. We present a Curry-style second-order type system with union and intersection types for the lambda-calculus with constructors of Arbiser, Miquel and Rios, an extension of lambda-calculus with a pattern matching mechanism for variadic constructors. We then prove the strong normalisation and the absence of match failure for a restriction of this system, by adapting the standard reducibility method.

Introduction

Pattern matching is a crucial feature in modern programming languages. It appeared in the late 60's [11], first as a simple detection of rigidly specified values. Although it still has this basic form in most imperative languages (as the `case` of Pascal or the `switch` of C), it now comes with more elaborated features in main functional programming languages [17, 12, 16] and proof assistants (especially those based on type theory [6, 1]). In particular, the pattern matching "à la ML" is able to decompose complex data-structures.

From the theoretical point of view, many approaches have been proposed to extend lambda-calculus [4] with pattern matching facilities, such as the *Rho-calculus* [8], the *Pure pattern calculus* [15] and the *Lambda calculus with constructors* [2]. Typed versions have also been presented for such calculi [5, 13, 19, 14].

The lambda-calculus with constructors [3] decomposes the pattern matching *à la ML* using a case construct

$$\{\!|c_1 \mapsto u_1; \ldots; c_n \mapsto u_n|\!\} \cdot t$$

performing case analysis on constant constructors, in the spirit of the `case` of Pascal. Composite data structures consist of constructor applied to one or many arguments. Their destruction is achieved using a commutation rule between case and application[1]:

$$(\textsc{CaseApp}) \qquad \{\!|\theta|\!\} \cdot (tu) \;=\; (\{\!|\theta|\!\} \cdot t)u$$

1998 ACM Subject Classification: F.3.2, F.4.3.

Key words and phrases: lambda-calculus, polymorphism, pattern matching, strong normalisation, reducibility candidates.

[1]Which differs from the commutative conversion rules [10] coming from logic.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-7 (1:2) 2010

Thanks to this rule, one can encode the whole ML-style pattern matching in the calculus, and write destruction functions on more complex data types, such as for instance the predecessor function: $pred = \lambda x.\{\!|\mathtt{0} \mapsto \mathtt{0}; \mathtt{S} \mapsto \lambda z.z|\!\} \cdot x$,
which satisfies:
$$\begin{aligned} pred\ (\mathtt{S}\ n) &= \{\!|\mathtt{0} \mapsto \mathtt{0}; \mathtt{S} \mapsto \lambda z.z|\!\} \cdot (\mathtt{S}\ n) \\ &= (\{\!|\mathtt{0} \mapsto \mathtt{0}; \mathtt{S} \mapsto \lambda z.z|\!\} \cdot \mathtt{S})\ n \\ &= (\lambda z.z)\ n \\ &= n \end{aligned}$$

Actually, one can even encode pattern matching for variadic constructors. The λ-calculus with constructors enjoys many good properties, such as confluence and separation (in the spirit of Böhm's theorem). It comprises nine rules, among which we can distinguish *essential rules* —such as β-reduction, case analysis and CaseApp— that are necessary to reduce terms to values, and *unessential rules* —like η-reduction— whose main role is to guarantee confluence and separation properties.

A polymorphic type system has been proposed for this calculus in [19], thus addressing the problem of typing the case construct in presence of the CaseApp commutation rule. This paper is an extended version of [19] with major changes, since some results appear to be incorrect (*cf.* Part 3). Indeed, typed lambda-calculus with constructors supports some non-terminating reductions, and also match failure can occur. This is due to one of the unessential rule: the composition between case constructions.

In this paper we drop out this composition rule from the calculus[2], and then justify this with realisability arguments. A semantic analysis using reducibility candidates ensures the strong normalisation of this restricted calculus. The main difficulty is to design a good notion of reducibility candidates which is able to cope with the commutation rule attached to the case. For that we introduce the notion of *case commutation normal form*, and we consider the usual reducibility candidates [10] up to case commutation. From this construction we deduce the main property of the typed calculus, including the absence of match failure for well typed terms.

Outline: Parts 1 and 2 respectively present the $\lambda_{\mathcal{C}}$-calculus and the type system. Part 3 is a discussion about the type system and the different reduction rules, and Part 4 the reducibility candidates model. Finally, Part 5 concludes with the main properties of the typed $\lambda_{\mathcal{C}}$-calculus.

1. The lambda-calculus with constructors

1.1. **Its syntax.** The syntax of the λ-calculus with constructors [3] is defined from two disjoint sets of symbols: *variables* (notation: x, y, z, etc.) and *constructors* (notation: $\mathtt{c}$, $\mathtt{d}$, etc. in typewriter font). It consists of two syntactic categories defined by mutual induction in Fig. 1: terms (notation: s, t, u, etc.) and case bindings (notation: θ, ϕ).

Terms include all the syntactic constructs of the λ-calculus, plus constructors (as constants) with a case construct (similar to the case construct of Pascal) to analyse them. There is also a constant ✠ (the *Daimon*, inherited from ludics [9]) representing immediate termination. It cannot appear in a term during reduction, but we keep it in the calculus for

[2]Losing thereby the separation property.

$Terms:$	s,t,u	$\triangleq$	$x \mid \lambda x.t \mid tu$	(λ-calculus)
		$\mid$	c	(Constructor)
		$\mid$	$\{\!\{\theta\}\!\} \cdot t$	(Case Construct)
		$\mid$	✠	(Daimon)
$CaseBindings:$	θ,ϕ	$\triangleq$	$\{\mathsf{c}_1 \mapsto u_1; \ldots; \mathsf{c}_n \mapsto u_n\}$ $\mathsf{c}_i \neq \mathsf{c}_j$ for $i \neq j$	(Case Binding)

Figure 1: $\lambda_{\mathcal{C}}$-terms and case bindings.

technical reasons (explained in Section 4.2). Case bindings are finite functions from constructors to terms. In order to ease the reading, we may write $\{\!\{\mathsf{c}_1 \mapsto u_1 \ ; \ \ldots \ ; \ \mathsf{c}_n \mapsto u_n\}\!\} \cdot t$ for $\{\!\{\{\mathsf{c}_1 \mapsto u_1; \ldots; \mathsf{c}_n \mapsto u_n\}\}\!\} \cdot t$.

Free and bound (occurrences of) variables are defined as usual, taking care that constructors are not variables and thus not subject to α-conversion. The set of free variables (denoted by $\mathcal{FV}(-)$) is defined for the new constructs by

$$\mathcal{FV}(\mathsf{c}) = \emptyset \qquad \mathcal{FV}(\{\!\{\theta\}\!\} \cdot t) = \mathcal{FV}(\theta) \cup \mathcal{FV}(t) \qquad \mathcal{FV}(\theta) = \cup_{(\mathsf{c} \mapsto u) \in \theta} \mathcal{FV}(u)$$

A term is *closed* when it has no free variable, and we write Λ_0 for the set of closed $\lambda_{\mathcal{C}}$-terms.

The usual operation of substitution on terms (notation: $t[x := u]$) is defined as expected, taking care of renaming bound variables when needed in order to prevent variable capture. Substitution on case bindings (notation: $\theta[x := u]$) is defined component-wise.

1.2. **Its operational semantics.** The reduction of $\lambda_{\mathcal{C}}$-calculus is based on the nine reduction rules given in Fig. 2 among which one can find the β and η reduction rules of the λ-calculus, now called AppLam and LamApp [3], respectively. We write $\to$ the contextual closure of these rules, and $\to^=$ (*resp.* $\to^+$, *resp.* $\to^*$) denotes its reflexive (*resp.* transitive, *resp.* reflexive and transitive) closure.

Case bindings behave like functions with finite domain. Therefore we may use the usual functional vocabulary: if $\theta = \{c_i \mapsto u_i \,/\, 1 \leq i \leq n\}$, then the *domain* of θ is the set $dom(\theta) = \{c_1, \ldots c_n\}$; also θ_c denotes u when $c \mapsto u \in \theta$. Case constructs are propagated through terms via the CaseApp, CaseLam and CaseCase commutation rules, and ultimately destructed with CaseCons reduction. For an explanation of the role and expressiveness of these rules, see [3].

The confluence or non confluence is known for every combination of the 9 reduction rules ([3] Theorem 1), and the full calculus is confluent. In this paper, we shall only consider the following sub-calculi, which are all confluent:

- $\lambda_{\mathcal{C}}^-$ denotes $\lambda_{\mathcal{C}}$-calculus with all the rules except CaseCase. In this paper we show that types ensure the strong normalisation of this calculus.
- $\lambda_{\mathtt{com}}$ is the calculus of case commutation (whose only rules are CaseApp and CaseLam). For technical reasons (*cf.* Part 4) we sometimes consider terms up to case commutation equivalence.

[3] In $\lambda_{\mathcal{C}}$-calculus, the name of each reduction rule consists of the names of the two constructions interacting for the reduction.

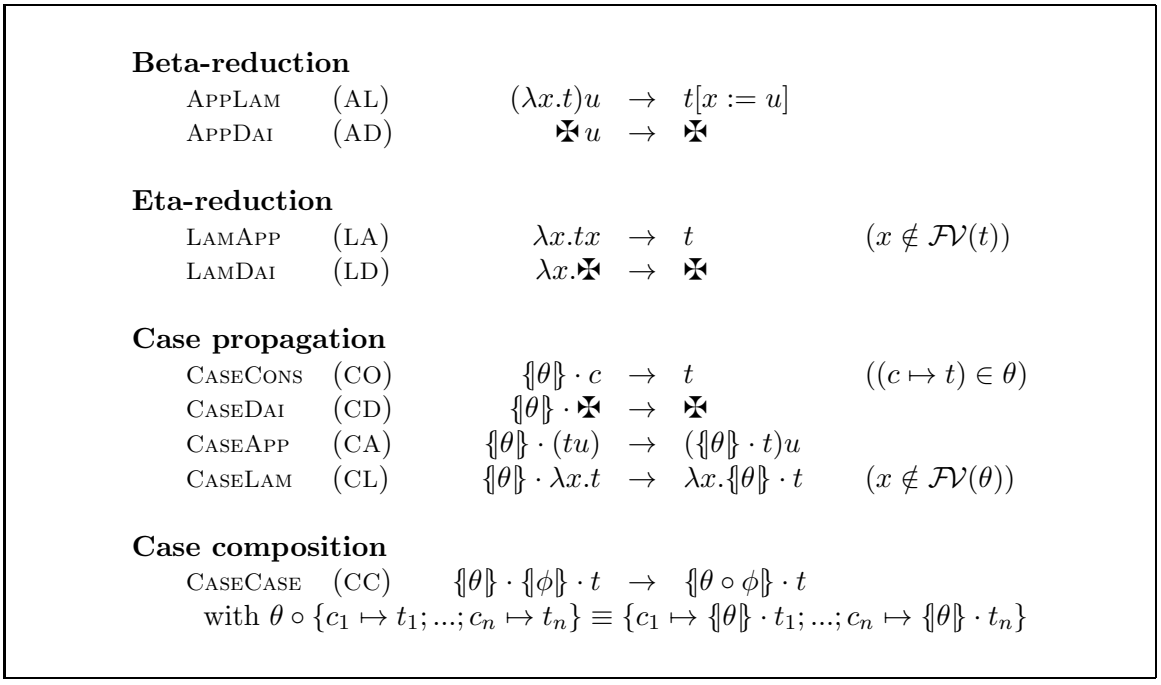

Figure 2: Reduction rules for $\lambda_{\mathcal{C}}$.

- $\lambda_{\mathcal{B}}$ is the complement calculus of $\lambda_{\mathtt{com}}$ in $\lambda_{\mathcal{C}}^{-}$: it is composed of rules AppLam, AppDai and LamApp, LamDai, CaseCons and CaseDai.

A term with no infinite reduction is said to be *strongly normalising*. By extension, a calculus is strongly normalising when all its terms are. It is also known that the whole calculus without AppLam is strongly normalising ([3], Proposition 2).

1.3. **Values in lambda-calculus with constructors.** In pure lambda-calculus, a value is a function (*i.e.* a λ-abstraction). In $\lambda_{\mathcal{C}}$ we call *data structure* a term of the form $\mathtt{c}\, t_1 \dots t_k$ where $\mathtt{c}$ is a constructor and $t_1, \dots, t_k$ $(k \geq 0)$ are arbitrary terms. We then call a *value* a term which is a λ-abstraction or a data structure. The set of values is written $\mathcal{V}$.

We say that a term is *defined* when it has no sub-term of the form $\{\!|\theta|\!\} \cdot \mathtt{c}$, with $\mathtt{c} \notin dom(\theta)$, and that it is *hereditarily defined* when all its reducts (in any number of steps) are defined. (Intuitively, non-defined terms contain pattern matching failures and therefore will be rejected by the type system.)

Proposition 1.1. *Every defined closed normal term is either* $\maltese$ *or a value.*

Proof. Let t be a closed defined term in normal form. By induction on the structure of t, we show that t is either $\maltese$ or $\lambda x.t_0$ or $\mathtt{c}t_1 \dots t_k$ for some constructor $\mathtt{c}$, 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 also is closed, defined and normal. 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. $\lambda_{\mathcal{C}}^{-}$.

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

Proof. First recall that $t \to t'$ implies $t[x := u] \to t'[x := u]$ ([3] Lemma 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 $\{\!|\theta|\!\} \cdot c$ with $c \notin dom(\theta)$, and this property is kept by replacing some sub-terms by x. So t also is defined. By induction on the reduction of t, we can easily conclude that if $t[x := u]$ is hereditarily defined, so is t. □

2. Type system

2.1. **An informal presentation.** The type system we want to define 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. To achieve polymorphism, we introduce type variables (written X, Y etc.) and universal type quantification (notation: $\forall X.T$). Instantiation is performed via a sub-typing judgement containing all the rules of system F with sub-typing such as presented in [18].

To type-check data structures, we associate to every constructor $\mathtt{c}$ a type constant $\mathbf{c}$ —written with bold font. We introduce a *type application* DT for applied structures, so that we can derive $\mathtt{c}\ \overrightarrow{t} : \mathbf{c}\ \overrightarrow{T}$ from $\overrightarrow{t} : \overrightarrow{T}$ (see 2.2 for more details on vectorial notations). Nevertheless, the formation of application types has to be restricted. Indeed, with a typing rule such as

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

if t is a term of type $bool \to U$, and u a term of type nat, we would be able to type term tu with type $(bool \to U)\, nat$, which may be a nonsense if t implements a function expecting only booleans. Furthermore, it would also enable typing non normalising terms like $\delta\delta$, as $\delta = \lambda x.xx$ is typable in system F.

For that reason we distinguish a sub-class of *data types* (notation: D, E). They will be the only types on the left-hand side of a type application. In practice this sub-class excludes arrow types and type variables (which could be instantiated by arbitrary types). To still keep the ability to quantify over data types, we introduce *data type variables* (notation: α, β etc.) and data type quantification.

To encode algebraic types, we add union types. For example, we could define a type of natural numbers with the equation $nat \equiv \mathbf{0} \cup \mathbf{S}\,(nat)$ (where $\mathtt{0}$ and $\mathtt{S}$ are constructors)[4].

[4]This would require a fixpoint operator, or a double sub-typing rule.

To distribute arrow among union, we also need *intersection types*:

$$(\mathbf{0} \cup \mathbf{S}(\mathit{nat})) \to T \equiv (\mathbf{0} \to T) \cap (\mathbf{S}(\mathit{nat}) \to T).$$

By symmetry, we add the existential quantifier.

Types :	$T, U :=$	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)
Data Types :	$D, E :=$	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 3: Types of $\lambda_{\mathcal{C}}$.

2.2. **The formal system.** We define a polymorphic type system with union and intersection for both terms and case bindings of $\lambda_{\mathcal{C}}$ (Fig. 3). It uses two spaces of type variables: *ordinary type variables* and *data type variables*. There are also two kinds of types: *ordinary types*, and their syntactic sub-class of *data types*.

In the following, ν denotes a variable which can be an ordinary type variable or a data type variable. The set $\mathcal{TV}(T)$ denotes the set of all free type variables of a type T:

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

We also use a vectorial notation for type application and arrow types:

$$\overrightarrow{T} \;:=\; [\,] \mid \overrightarrow{T}; T$$

$$\begin{array}{rcrcl}
\mathbf{c}[\,] & = & \mathbf{c} & \qquad \mathbf{c}(\overrightarrow{T}; T) & = & (\mathbf{c}\overrightarrow{T})T \\
[\,] \to U & = & U & \qquad (\overrightarrow{T}; T) \to U & = & \overrightarrow{T} \to (T \to U)
\end{array}$$

Typing rules (Fig. 4) 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. 5).

Case Binding: If $\theta = \{\mathtt{c}_i \mapsto u_i \,/\, 1 \leq i \leq n\}$ with $n \geq 0$.

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

Terms:

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

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

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

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

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

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

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

Figure 4: Typing rules

Type application takes precedence over all the other operators and is left associative. Sub-typing rule `Data` allows typing constructors with non-fixed arity:

$$\mathbf{c}T_1 \dots T_k \ \preccurlyeq\ T_{k+1} \to \mathbf{c}T_1 \dots T_k T_{k+1},$$

implies that if $\mathtt{c}t_1 \dots t_k$ has type $\mathbf{c}T_1 \dots T_k$, and if t_{k+1} has type T_{k+1}, then $\mathtt{c}t_1 \dots t_{k+1}$ has type $\mathbf{c}T_1 \dots T_{k+1}$. By iterating, we immediately get

$$(\Gamma \vdash t_i : T_i)_{i=1}^{n} \qquad \Longrightarrow \qquad \Gamma \vdash \mathtt{c}t_1 \dots t_n :\ \mathbf{c}T_1 \dots T_n$$

Having such variadic constructors allows for example to add or remove an element in an array locally (Example 2.1).

2.3. **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.

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

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

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

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

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

$$\forall\texttt{intro}\frac{T \preccurlyeq U}{T \preccurlyeq \forall\nu.U}{}^{\nu\notin\mathcal{TV}(T)} \qquad \forall\texttt{elim}\frac{-}{\forall X.T \preccurlyeq T\{X \leftarrow U\}} \qquad \forall\texttt{elimD}\frac{-}{\forall\alpha.T \preccurlyeq T\{\alpha \leftarrow D\}}$$

$$\exists\texttt{intro}\frac{-}{T\{X \leftarrow U\} \preccurlyeq \exists X.T} \qquad \exists\texttt{introD}\frac{-}{T\{\alpha \leftarrow D\} \preccurlyeq \exists\alpha.T} \qquad \exists\texttt{elim}\frac{U \preccurlyeq T}{\exists\nu.U \preccurlyeq T}{}^{\nu\notin\mathcal{TV}(T)}$$

$$\texttt{Data}\frac{-}{D \preccurlyeq T \to DT} \qquad \texttt{Constr}\frac{-}{\mathbf{c}_1\overrightarrow{T} \cap \mathbf{c}_2\overrightarrow{U} \preccurlyeq \forall\alpha.\alpha}{}^{\mathtt{c}_1\neq\mathtt{c}_2}$$

$$\texttt{App}/\cap\frac{-}{\bigcap_i D_iT_i \preccurlyeq (\bigcap_i D_i)(\bigcap_i T_i)} \qquad \texttt{App}/\forall\frac{-}{\forall\nu.(DT) \preccurlyeq (\forall\nu.D)(\forall\nu.T)}$$

$$\to/\cap\frac{-}{\bigcap_i T_i \to U_i \preccurlyeq (\bigcap_i T_i) \to (\bigcap_i U_i)} \qquad \to/\forall\frac{-}{\forall\nu.(T \to U) \preccurlyeq (\forall\nu.T) \to (\forall\nu.U)}$$

$$\to/\cup\frac{-}{\bigcap_i T_i \to U_i \preccurlyeq (\bigcup_i T_i) \to (\bigcup_i U_i)} \qquad \to/\exists\frac{-}{\forall\nu.(T \to U) \preccurlyeq (\exists\nu.T) \to (\exists\nu.U)}$$

$$\cup/\texttt{AppR}\frac{-}{D(\bigcup_i T_i) \preccurlyeq \bigcup_i DT_i} \qquad \cup/\texttt{AppL}\frac{-}{(\bigcup_i D_i)T \preccurlyeq \bigcup_i (D_iT)}$$

$$\exists/\texttt{AppR}\frac{-}{D(\exists\nu.T) \preccurlyeq \exists\nu.DT}{}^{\nu\notin\mathcal{TV}(D)} \qquad \exists/\texttt{AppL}\frac{-}{(\exists\nu.D)T \preccurlyeq \exists\nu.DT}{}^{\nu\notin\mathcal{TV}(T)}$$

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

Figure 5: Sub-typing rules.

This is the point that allows CaseApp commutation rule to be well typed.

Example 2.1. Consider the constructor $\mathtt{c}_\diamond$ that initialises arrays. Then the case binding $\theta = \{\mathtt{c}_\diamond \mapsto \lambda xy.\mathtt{c}_\diamond x\}$ removes the second element of any array:

$$\{\!|\theta|\!\} \cdot (\mathtt{c}_\diamond t_1 t_2 t_3) \ \to^3_{\mathbf{CA}} \ (\{\!|\theta|\!\} \cdot \mathtt{c}_\diamond) t_1 t_2 t_3 \ \to \ (\lambda xy.\mathtt{c}_\diamond x) t_1 t_2 t_3 \ \to^3 \ \mathtt{c}_\diamond t_1 t_3$$

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

$$\dfrac{\dfrac{}{\vdash \lambda xy.\mathtt{c}_\diamond x : T_1 \to T_2 \to \mathbf{c}_\diamond T_1} \quad \dfrac{\dfrac{}{\mathbf{c}_\diamond T_1 \preccurlyeq T_3 \to \mathbf{c}_\diamond T_1 T_3}}{T_1 \to T_2 \to \mathbf{c}_\diamond T_1 \preccurlyeq T_1 \to T_2 \to T_3 \to \mathbf{c}_\diamond T_1 T_3}}{\dfrac{\vdash \lambda xy.\mathtt{c}_\diamond x : T_1 \to T_2 \to T_3 \to \mathbf{c}_\diamond T_1 T_3}{\vdash \theta : \mathbf{c}_\diamond T_1 T_2 T_3 \to \mathbf{c}_\diamond T_1 T_3}}$$

$$\dfrac{\dfrac{}{\vdash \theta : \mathbf{c}_\diamond T_1 T_2 T_3 \to \mathbf{c}_\diamond T_1 T_3} \quad \dfrac{\begin{array}{r}\vdash t_1 : T_1 \\ \vdash t_2 : T_2 \\ \vdash t_3 : T_3\end{array}}{\vdash \mathtt{c}_\diamond t_1 t_2 t_3 : \mathbf{c}_\diamond T_1 T_2 T_3}}{\vdash \{\!|\theta|\!\} \cdot (\mathtt{c}_\diamond t_1 t_2 t_3) : \mathbf{c}_\diamond T_1 T_3}$$

We can also give the same type to $(\{\!|\theta|\!\} \cdot \mathtt{c}_\diamond) t_1 t_2 t_3$ by choosing another possible type for θ (we write $\overrightarrow{T} = T_1; T_2; T_3$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \lambda xy.\mathtt{c}_\diamond x : \overrightarrow{T} \to \mathbf{c}_\diamond T_1 T_3}}{\vdash \theta : \mathbf{c}_\diamond \to \overrightarrow{T} \to \mathbf{c}_\diamond T_1 T_3} \quad \dfrac{}{\vdash \mathtt{c}_\diamond : \mathbf{c}_\diamond}}{\vdash \{\!|\theta|\!\} \cdot \mathtt{c}_\diamond : \overrightarrow{T} \to \mathbf{c}_\diamond T_1 T_3} \quad \begin{array}{r}\vdash t_1 : T_1 \\ \vdash t_2 : T_2 \\ \vdash t_3 : T_3\end{array}}{\vdash (\{\!|\theta|\!\} \cdot \mathtt{c}_\diamond) t_1 t_2 t_3 : \mathbf{c}_\diamond T_1 T_3}$$

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 numbers of arguments. This make 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:

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

If the case binding includes many branches, we can either chose one of them, or give to it an intersection type, and then commute intersection with arrow.

Example 2.2. Assume *nat* is a type satisfying $\mathit{nat} \equiv \mathbf{0} \cup \mathbf{S}\,\mathit{nat}$. The predecessor case bindings $\theta = \{\mathtt{0} \mapsto \mathtt{0}\ ; \mathtt{S} \mapsto \lambda x.x\}$ has both types $\mathbf{0} \to \mathit{nat}$ and $\mathbf{S}\,\mathit{nat} \to \mathit{nat}$. Hence we can derive

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

and thus θ has type $\mathit{nat} \to \mathit{nat}$.

The rule $\mathbf{Cb}_\perp$ is a kind of generalisation of this typing derivation: indeed, if $\theta = \{\mathtt{c}_i \mapsto u_i \,/\, 1 \leq i \leq n\}$, with $\vdash u_i : \overrightarrow{U_i} \to T_i$, then for any $J \subseteq [1..n]$, the judgement $\vdash \theta : \bigcup_{i\in J} \mathtt{c}_i\overrightarrow{U_i} \to \bigcup_{i\in J} T_i$ 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, $\mathbf{Cb}_\perp$ 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 Proposition 1.1 and Remark 5.8).

3. Restricted lambda calculus with constructor

The type system described in the previous section is the one presented in [19]. It appears that the final result (Proposition 15) of that paper is wrong[5]. Here we present a simple counterexample, and we explain how we cope with the problem.

3.1. The problem of case-composition.

Typed $\lambda_{\mathcal{C}}$-calculus does not prevent match failure. Indeed, the CaseCase rule can create sub-terms whose typing is not checked in the "dead branches" of a case-binding. For instance, if

$$\phi = \{\mathtt{d} \mapsto \mathtt{d'}\} \quad \text{and} \quad \theta = \{\mathtt{c} \mapsto \mathtt{d}\,;\mathtt{c'} \mapsto \mathtt{c'}\},$$

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

So we can derive $\vdash \{\!|\theta|\!\} \cdot \mathtt{c} : \mathbf{d}$ and then $\vdash \{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot \mathtt{c} : \mathbf{d'}$. This makes sense because we can obtain $\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot \mathtt{c} \to^* \mathtt{d'}$ by applying twice the rule CaseCons. In θ, $\mathtt{c'} \mapsto \mathtt{c'}$ is a dead branch and is forgotten by the typing (once we know that $\mathtt{c'}$ itself is typable). However, we can also apply the rule CaseCase and get $\{\!|\phi \circ \theta|\!\} \cdot \mathtt{c}$. Hence, the second branch of the case-binding is $\mathtt{c'} \mapsto \{\!|\phi|\!\} \cdot \mathtt{c'}$, which raises a match failure and is hardly 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 $\{\!|\phi|\!\} \cdot \{\!|\theta|\!\} \cdot \mathtt{c'}$, that reduces on the same match-failing term $\{\!|\phi|\!\} \cdot \mathtt{c'}$). But yet the CaseCase rule can create redices in branches that have been dropped by the typing.

Actually, the situation is even worse. Rule CaseCase, together with the other rules, makes some typable terms non-terminating:

Let $\phi = \{\mathtt{d} \mapsto \delta\}$ and $\theta = \{\mathtt{c} \mapsto \mathtt{d}\,;\mathtt{c'} \mapsto \mathtt{d}\delta\}$, where $\delta = \lambda x.xx$. Then we can derive

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

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 CaseCase rule, but with it we can reduce

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

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

[5] In [19] the proof fails at Lemma 10. There is a counterexample to the converse of equivalence (13), surprisingly due to the notion of modified substitution used there.

3.2. **Restriction of the calculus.** Remember that $\lambda_{\mathcal{C}}^{-}$, *i.e.*, the $\lambda_{\mathcal{C}}$-calculus without the rule CaseCase, is confluent (*cf.* Part 1). We will see in Part 5 that typed $\lambda_{\mathcal{C}}^{-}$-calculus enjoys the perfect normalisation property.

Actually, rule CaseCase was introduced in the lambda calculus with constructors in order to satisfy the separation property ([3], Theorem 2) —and same as for the rule LamApp, the usual eta-reduction. But it is unessential for computing in the lambda calculus with constructors (cf. the discussion in Section 5.3).

Also from now on we remove the case composition from the calculus, and we consider the $\lambda_{\mathcal{C}}^{-}$-calculus. In particular, we now use notation $\to$ for $\to_{\lambda_{\mathcal{C}}^{-}}$.

The set of terms is kept unchanged, so we use the same definition of *defined term* and of *value* as in $\lambda_{\mathcal{C}}$-calculus. Note that Proposition 1.1 still holds in $\lambda_{\mathcal{C}}^{-}$. The set of closed terms that are perfectly normalising for $\lambda_{\mathcal{C}}^{-}$ rules is denoted by PN_0. By extension we say that a case binding θ is in PN_0 when it is composed of closed and perfectly normalising terms for $\lambda_{\mathcal{C}}^{-}$.

In the following, we prove the perfect normalisation (*i.e.* strong normalisation without match failure) of typed $\lambda_{\mathcal{C}}^{-}$-calculus.

4. Reducibility Candidates

Reducibility candidates [10] are sets of closed and perfectly normalising terms. They will later be used to interpret types. In this paper we complete their usual meaning with the notion of *data candidates*. In the following, we denote by $Red_n(t)$ the set of terms to which t reduces in n steps, by $Red_*(t)$ the union of all these sets for n in $\mathbb{N}$, and by $Red_+(t)$ the union for $n \geq 1$.

Because of their "ill-behaviour" w.r.t. typing, commutation rules will be treated with a special attention. Remember that we write $\to_{\mathrm{c}}$ the union of CaseApp and CaseLam, and $\lambda_{\mathtt{com}}$ denotes the calculus containing only these two rules. Conversely, the calculus consisting of all reduction rules of $\lambda_{\mathcal{C}}^{-}$ *except* CaseApp and CaseLam is written $\lambda_{\mathcal{B}}$ (and, as expected, $\to_{\mathcal{B}}$ denotes the union of AppLam, AppDai, LamApp, LamDai, CaseCons and CaseDai).

In this section, we first give some properties of $\lambda_{\mathtt{com}}$-normal forms. Next we give a definition of reducibility candidates and a method to construct them using closure operator. Then we emphasise the connection between reducibility candidates and values. Finally we define some operations on reducibility candidates.

4.1. **Case-commutation normal form.** The reduction system $\lambda_{\mathtt{com}}$ is strongly normalising. Indeed, reducing a term in $\lambda_{\mathtt{com}}$ decreases its *structural measure* s, introduced in [3] as follows:

$$\begin{array}{rl} s(x) & = s(c) \;=\; s(\maltese) \;=\; 1 \\ s(\lambda x.t) & = s(t) + 1 \\ s(tu) & = s(t) + s(u) \\ s(\{\!|\theta|\!\} \cdot t) & = s(t) \times (s(\theta) + 2) \\ s(\{c_i \mapsto u_i \,/\, 1 \leq i \leq n\}) & = \sum_{i=1}^{n} s(u_i) \end{array}$$

In the following, we will often need to consider terms up to case-commutation rules. The normal form of a term t for $\to_{\mathrm{c}}$ is written $\downarrow t$. It is characterised by the following

equations:

$$\begin{array}{rclcrcl}
\downarrow x & = & x & & \downarrow\{\!|\theta|\!\}\cdot x & = & \{\!|\downarrow\theta|\!\}\cdot x \\
\downarrow c & = & c & & \downarrow\{\!|\theta|\!\}\cdot c & = & \{\!|\downarrow\theta|\!\}\cdot c \\
\downarrow\maltese & = & \maltese & & \downarrow\{\!|\theta|\!\}\cdot\maltese & = & \{\!|\downarrow\theta|\!\}\cdot\maltese \\
\downarrow\lambda x.t & = & \lambda x.\downarrow t & & \downarrow\{\!|\theta|\!\}\cdot\lambda x.t & = & \lambda x.\downarrow(\{\!|\theta|\!\}\cdot t) \\
\downarrow(tu) & = & \downarrow t\,\downarrow u & & \downarrow\{\!|\theta|\!\}\cdot(tu) & = & \downarrow(\{\!|\theta|\!\}\cdot t)\,\downarrow u \\
\downarrow\{c_i\mapsto u_i\,/\,1\leq i\leq n\} & = & \{c_i\mapsto\downarrow u_i\,/\,1\leq i\leq n\} & & \downarrow\big(\{\!|\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t\big) & = & \downarrow(\{\!|\theta|\!\}\cdot\downarrow\{\!|\phi|\!\}\cdot t)
\end{array}$$

and by $\downarrow\big(\{\!|\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t\big) = \{\!|\downarrow\theta|\!\}\cdot\{\!|\phi|\!\}\cdot t$ if $\downarrow\{\!|\phi|\!\}\cdot t \;=\; \{\!|\phi|\!\}\cdot t$.

To deal with perfect normalisation, we can consider terms up to case commutation, since both well-definition and strong normalisation are preserved by $\lambda_{\mathtt{com}}$-reduction and expansion. That is what Corollary 4.3 expresses.

Lemma 4.1. *If $\downarrow t$ is defined, so is t.*

Lemma 4.2. *$t\to_{\mathcal{B}} t'$ implies $\downarrow t\to^{+}\downarrow t'$*

Proof. By induction on t.

- If $t=x$, $\maltese$ or $\mathtt{c}$, then t is not reducible.
- If $t=\lambda x.t_0$, then $t'=\lambda x.t_0'$ with $t_0\to_{\mathcal{B}}t_0'$ and we conclude by induction.
- If $t=t_1t_2$, three different cases can occur:
  - $t'=t_1t_2'$ or $t_1't_2$ with $t_i\to_{\mathcal{B}}t_i'$. Hence we conclude by induction
  - $t_1=\maltese$ and $t'=\maltese$. In that case $\downarrow t=(\maltese\,\downarrow t_2)$ reduces to $\maltese=\downarrow t'$.
  - $t_1=\lambda x.t_0$ 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) $\downarrow(t_0[x:=t_2])$.
- 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 $t_0=c$ and $c\mapsto u\in\theta$, or $t'=\maltese$ and $t_0=\maltese$. In both last cases, $\downarrow t=\{\!|\downarrow\theta|\!\}\cdot t_0\to\downarrow t'$. □

Corollary 4.3. *If $\downarrow t\in PN_0$, then $t\in PN_0$.*

Proof. First $u\in Red_*(t)$ implies $\downarrow u\in Red_*(\downarrow t)$ by Lemma 4.2. So Lemma 4.1 entails that all reducts of t are defined as soon as all reducts of $\downarrow t$ are.
Now assume there is an infinite reduction $t=t_0\to t_1\to t_2\dots$ Since $\to_{\mathrm{c}}$ is strongly normalising, this reduction chain contains an infinity of $\to_{\mathcal{B}}$ reduction steps: $t=t_0\overset{*}{\to}_{\mathrm{c}}t_{i_1}\to_{\mathcal{B}}t_{j_1}\overset{*}{\to}_{\mathrm{c}}t_{i_2}\to_{\mathcal{B}}t_{j_2}\dots$ So $\downarrow t_{j_k}=\downarrow t_{i_{k+1}}$ and $\downarrow t_{i_k}\to^{+}\downarrow t_{j_k}$ by Lemma 4.2. Hence there is an infinite reduction

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

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

4.2. **Definition of reducibility candidates.** The definition of reducibility candidates is founded on the notion of values and neutral terms. Recall that the set $\mathcal{V}$ of values includes all data structures and lambda-abstractions. We then call *neutral* the terms which are not values. The set of defined closed neutral terms is written $\mathcal{N}_D$. In particular, $\maltese$ is neutral.

Remark 4.4. Since $t\in\mathcal{V}$ implies $\downarrow t\in\mathcal{V}$, Lemma 4.1 leads to

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

A set S of closed terms is a *reducibility candidate* when it satisfies:
(**CR1**)**:** Perfect normalisation: $S \subseteq PN_0$
(**CR2**)**:** Stability by reduction: $t \in S \Rightarrow Red_1(t) \subseteq S$
(**CR3**)**:** Stability by neutral expansion: if $t \in \mathcal{N}_D$, then $Red_1(t) \subseteq S \;\Rightarrow\; t \in S$
(**CR4**)**:** Stability by case-commutation: if $t \to_{\mathtt{c}} t'$, and $t' \in S$ then $t \in S$

We denote by $\mathcal{CR}$ the set of all reducibility candidates, and by (**CR**) the conjunction of all four conditions. The usual stability properties for reducibility candidates are (**CR1**), (**CR2**) and (**CR3**). Property (**CR4**) is specific to this type system, and will be necessary in order to prove the validity of the **Cb** rule.

Note that every reducibility candidate is non empty (it contains ✠ as neutral term with no reduct). This will be important when interpreting arrow types. Moreover PN_0 is in $\mathcal{CR}$ (resulting from Corollary 4.3, PN_0 is stable by (**CR4**)).

In some of the proofs of this paper we need to use another definition of reducibility candidates, that is equivalent.

Lemma 4.5. *Given $S \subseteq \Lambda_0$, we define two new stability properties:*
(**CR2′**)**:** $t \in S \;\Rightarrow\; Red_*(t) \subseteq S$
(**CR4′**)**:** $\downarrow t \in S \;\Rightarrow\; t \in S$
Then a reducibility candidate can be characterised by (**CR1**)*,* (**CR2′**)*,* (**CR3**) *and* (**CR4′**) *since*

$$(\mathbf{CR2}) \;\Leftrightarrow (\mathbf{CR2'}) \tag{4.1}$$

$$(\mathbf{CR2}) \wedge (\mathbf{CR4}) \;\Leftrightarrow (\mathbf{CR2'}) \wedge (\mathbf{CR4'}) \tag{4.2}$$

Proof.

(4.1) (**CR2′**) obviously implies (**CR2**). Conversely, if S satisfies (**CR2**) and $t \in S$, then we can prove by induction on n that $t \to^n u$ implies $u \in S$.

(4.2) Assume S satisfies (**CR4**). If t is a term such that $\downarrow t \in S$, we can see by induction on the reduction $t \overset{*}{\to}_{\mathtt{c}} \downarrow t$ that $t \in S$. Conversely, if S satisfies (**CR2′**) and (**CR4′**), then for any $t' \in S$ and any $t \to_{\mathtt{c}} t'$, we have $\downarrow t = \downarrow t'$ is in S by (**CR2′**) (since $t' \to^* \downarrow t'$), thus $t \in S$ by (**CR4′**). □

4.3. **Closure properties.** A *non-expansed candidate* is a set of terms that satisfies (**CR1**) and (**CR2**). Sets that satisfy (**CR4**) in addition (or equivalently (**CR4′**)) are called *pre-candidates of reducibility*. We write $\mathcal{PCR}$ for the family of pre-candidates. For instance $\{\mathtt{c}\}$ is a pre-candidate for any constructor $\mathtt{c}$. We will see that such a pre-candidates can be closed by (**CR3**) to obtain a reducibility candidate.

Definition 4.6. For $X \subseteq \Lambda_0$, we note $\overline{X}$ its closure by (**CR3**). It is defined inductively by

$$\frac{t \in X}{t \in \overline{X}} \qquad \frac{t \in \mathcal{N}_D \quad Red_1(t) \subseteq \overline{X}}{t \in \overline{X}}$$

Lemma 4.7. *If $P \in \mathcal{PCR}$, then $\overline{P}$ is the smallest reducibility candidate containing P.*

Proof. $\overline{P}$ satisfies (**CR3**) by definition. Using the inductive definition, it is immediate to check (by induction) that it satisfies (**CR1**) and (**CR2′**). Now we prove by induction that it satisfies (**CR4′**). Let $t \in \Lambda_0$ such that $\downarrow t \in \overline{P}$.

- If $\downarrow t \in P$ then $t \in P$ since $P \in \mathcal{PCR}$ and thus satisfies (**CR4′**).

- Else $\downarrow t \in \mathcal{N}_D$ and $Red_1(\downarrow t) \subseteq \overline{P}$. In that case, t also is in $\mathcal{N}_D$ (Remark 4.4) and for all $u \in Red_1(t)$, $\downarrow u \in Red_*(\downarrow t)$ (by Lemma 4.2). Moreover, $Red_*(\downarrow t) \subseteq \overline{P}$ by (**CR2′**), thus $\downarrow u \in \overline{P}$. By induction hypothesis, it implies that $u \in \overline{P}$. Hence $Red_1(t) \subseteq \overline{P}$, so $t \in \overline{P}$ for being neutral.

Finally $\overline{P}$ is a reducibility candidate. Moreover, if S in $\mathcal{CR}$ contains P, it also contains $\overline{P}$ by (**CR3**). □

In the previous lemma it would not be sufficient to assume that P is a non-expansed candidate, to conclude $\overline{P} \in \mathcal{CR}$ (see example below). We later (in Lemma 4.15) characterise more precisely when a non-expansed candidate can be closed to obtain a reducibility candidate.

Example 4.8. Let $t = \lambda y.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot y$ and $u = \{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda y.y$. Then $u \to_{\mathrm{c}} t$.
The set $S = \{\lambda x.t\}$ satisfies (**CR1**) and (**CR2**) but $\overline{S}$ does not satisfy (**CR4**) since $\lambda x.u \notin \overline{S}$. So $\overline{S}$ is not a reducibility candidate.

Stability under (**CR3**) also entails that every reducibility candidate is infinite: if $\mathcal{A}$ is a reducibility candidate containing 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 increasing family of terms of $\mathcal{A}$.

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

Remark 4.9. Since the closure by (**CR3**) only adds neutral terms, if P is a pre-candidate whose all values are data-structures, then $\overline{P} \in \mathcal{DC}$. In particular $\overline{\{\mathsf{c}\}}$ is a data candidate for any constructor c.

4.4. **Reducibility Candidates and values.** 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 $\mathcal{V}al(t)$ (*resp.* $\mathcal{V}al(S)$), the set of values to which t (*resp.* a term of S) reduces:

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

Note that, $\mathcal{V}$ being closed by reduction, $\mathcal{V}al(S)$ is a non-expansed candidate for any set S of perfectly normalising terms. However, it is not necessarily a pre-candidate. Indeed, even if $\mathcal{A} \in \mathcal{CR}$ it does not insure $\mathcal{V}al(\mathcal{A}) \in \mathcal{PCR}$.

Example 4.10. Consider the reducibility candidate $\overline{S}$, with

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

$\mathcal{V}al(\overline{S})$ is not stable under (**CR4**) since it does not contain $\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda x.x$ whereas $\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot \lambda x.x \quad \to_{\mathrm{c}} \quad \lambda x.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot x$ and $\lambda x.\{\!|\mathsf{c} \mapsto \mathsf{c}|\!\} \cdot x \in \mathcal{V}al(\overline{S})$.

Also it is generally not possible to use the closure operator on a set of values $\mathcal{V}al(S)$ to construct a reducibility candidate. However, the values of a reducibility candidate are, in some extent, sufficient to define it (Corollary 4.12).

Lemma 4.11. *If $t \in PN_0$ and $\mathcal{A} \in \mathcal{CR}$, then* $\quad t \in \mathcal{A} \ \Leftrightarrow\ \mathcal{V}al(t) \subseteq \mathcal{A}$.

Proof. The implication is obvious using (**CR2′**).
We prove the converse by induction on the reduction of t (that is well-founded for strongly normalising terms). Assume $\mathcal{V}al(t) \subseteq \mathcal{A}$ and prove that $t \in \mathcal{A}$. If t is a value it is clear since $t \in \mathcal{V}al(t)$. Otherwise $t \in \mathcal{N}_D$, and for all u in $Red_1(t)$, $u \in \mathcal{A}$ by induction hypothesis (since $\mathcal{V}al(u) \subseteq \mathcal{V}al(t) \subseteq \mathcal{A}$). So $t \in \mathcal{A}$ by (**CR3**). □

Corollary 4.12. *Let $\mathcal{A}, \mathcal{B} \in \mathcal{CR}$. Then $\mathcal{V}al(\mathcal{A}) = \mathcal{V}al(\mathcal{B})$ 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 $\mathcal{V}al(\mathcal{A}) = \mathcal{V}al(\mathcal{B})$. By Lemma 4.11,

$$\begin{array}{l} t \in \mathcal{A} \textit{ iff } \mathcal{V}al(t) \subseteq \mathcal{A} \\ \qquad \textit{iff } \mathcal{V}al(t) \subseteq \mathcal{V}al(\mathcal{A}) \\ \qquad \textit{iff } \mathcal{V}al(t) \subseteq \mathcal{V}al(\mathcal{B}) \\ \qquad \textit{iff } \mathcal{V}al(t) \subseteq \mathcal{B} \\ \qquad \textit{iff } t \in \mathcal{B} \end{array}$$ □

This characterisation of a reducibility candidate by its values will be used in the next section to prove that our class $\mathcal{CR}$ is stable under union. For that, we also use a sufficient condition described in [20]: the *principal reduct* property.

Lemma 4.13. *Every $t \in \mathcal{N}_D$ has a reduct (in one step) $u \in \Lambda_0$ 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 define inductively, for every $t \in \mathcal{N}_D$ *that can reduce on a value*, a term $p(t)$:

$$\begin{array}{rcl} p((\lambda x.t_0)t_1 \dots t_k) & = & t_0[x := t_1]\ t_2 \dots t_k \\ p((\{\!|\theta|\!\} \cdot t_0)t_1 \dots t_k) & = & p(\{\!|\theta|\!\} \cdot t_0)t_1 \dots t_k \\ p(\{\!|\theta|\!\} \cdot c) & = & u \quad \text{if } c \mapsto u \in \theta \\ p(\{\!|\theta|\!\} \cdot \lambda x.t') & = & \lambda x.\{\!|\theta|\!\} \cdot t' \\ p(\{\!|\theta|\!\} \cdot t_1 t_2) & = & (\{\!|\theta|\!\} \cdot t_1)t_2 \\ p(\{\!|\theta|\!\} \cdot \{\!|\phi|\!\} \cdot t') & = & \{\!|\theta|\!\} \cdot p(\{\!|\phi|\!\} \cdot t') \end{array}$$

The point is that when a neutral term reduces on a value, it is necessarily by a reduction step performed at the root of the term (a so-called *head reduction*). The term $p(t)$ is obtained from t by reducing in head position. Every reduction chain leading from t to a value v begins eventually with reductions in sub-terms, and then the head-reduction is performed and gives a term u', that reduces on (or is) v. So to go from t to u' we can first reduce in head position and get $p(t)$, and then perform the same reductions in the sub-terms to get u'. □

4.5. **Candidates operators.** Since we aim to interpret types by reducibility candidates, we need to define all type operations in $\mathcal{CR}$. The definition of arrow is standard [10]. Here we also define the set application: for $\mathcal{A}, \mathcal{B} \subseteq \Lambda_0$,

$$\begin{array}{rcl} \mathcal{A} \to \mathcal{B} & \triangleq & \{t \in \Lambda_0 \ / \ \forall u \in \mathcal{A},\ tu \in \mathcal{B}\} \\ \mathcal{A}\mathcal{B} & \triangleq & \{tu \ / \ t \in \mathcal{A},\ u \in \mathcal{B}\} \end{array}$$

It is standard that $\mathcal{CR}$ is stable under arrow (we prove it in Lemma 4.16), as soon as candidates are not empty (that is the case here, since they all contain ✠). On the other hand, there is no reason for $\mathcal{CR}$ to be closed under application. Indeed, none of (**CR1**),

(**CR2**), (**CR3**) and (**CR4**) is preserved by application. In Lemma 4.16 (4.6) we see a way to construct a reducibility candidate by applying candidate to an other one. The family $\mathcal{CR}$ is naturally closed by intersection. We use the same method as in [20, Corollary 4.12] to deduce its stability under union (4.4).

Lemma 4.14. *For any family* $(P_i \in \mathcal{PCR})_{i\in I}$, $\overline{\bigcup P_i} \subseteq \bigcup \overline{P}_i$.

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

- If $t \in \bigcup P_i$, then there is $j \in I$ such that $t \in P_j \subseteq \overline{P_j}$
- If $t \in \mathcal{N}_D$ and $Red_1(t) \subseteq \overline{\bigcup P_i}$, let u be a principal reduct of t. Then $\mathcal{V}al(t) = \mathcal{V}al(u)$ (Lemma 4.13). Since $u \in Red_1(t)$, $u \in \overline{P_j}$ for some j by induction hypothesis. So $\mathcal{V}al(u) \subseteq \overline{P_j}$ by (**CR2**), and using Lemma 4.11 we get $t \in \overline{P_j}$. □

Lemma 4.15. *Let S be a non-expansed candidate. Then $\overline{S}$ is a reducibility candidate if, for any $t, t' \in \Lambda_0$,*

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

Proof. By definition $\overline{S}$ satisfies (**CR3**). The closure operator $\overline{\cdot}$ preserves (**CR1**) and (**CR2**), so these two properties also hold in $\overline{S}$. Now, we need to prove (**CR4′**). Let $\downarrow t \in \overline{S}$. By Corollary 4.3, $\downarrow t \in PN_0$ implies $t \in PN_0$. We prove by induction on its reduction that $t \in \overline{S}$. If $t = \downarrow t$ it is clear; else let t' such that $t \to_c t' \overset{*}{\to}_c \downarrow t$. By induction hypothesis, $t' \in \overline{S}$.

- If $t' \in S$ then by hypothesis $t \in \overline{S}$.
- Otherwise $t' \in \mathcal{N}_D$ and $Red_1(t') \in \overline{S}$ (by definition of the closure operator). Hence t also is in $\mathcal{N}_D$ (same as Remark 4.4). Moreover, for any $u \in Red_1(t)$, $\downarrow t \to^* \downarrow u$ by Lemma 4.2. So $\downarrow u \in \overline{S}$ by (**CR2**), and $u \in \overline{S}$ by induction hypothesis. Thus $Red_1(t) \subseteq \overline{S}$ and $t \in \overline{S}$.

So $\overline{S}$ also satisfies (**CR4′**), it is then a reducibility candidate. □

Lemma 4.16. *Given $(\mathcal{A}_i)$ and $(\mathcal{D}_i)$ families (possibly infinite) of $\mathcal{CR}$ and $\mathcal{DC}$ respectively, $\mathcal{A} \in \mathcal{CR}$, $\mathcal{D} \in \mathcal{DC}$, and S a non-expansed candidate that is non-empty,*

$$\bigcap \mathcal{A}_i \in \mathcal{CR} \qquad \text{and} \qquad \bigcap \mathcal{D}_i \in \mathcal{DC} \tag{4.3}$$

$$\bigcup \mathcal{A}_i \in \mathcal{CR} \qquad \text{and} \qquad \bigcup \mathcal{D}_i \in \mathcal{DC} \tag{4.4}$$

$$S \to \mathcal{A} \ \in \ \mathcal{CR} \tag{4.5}$$

$$\overline{\mathcal{D}\mathcal{A}} \ \in \ \mathcal{DC} \tag{4.6}$$

Proof.

(4.3) (**CR1**), (**CR2**), (**CR3**) and (**CR4**) are each preserved by intersection, so $\bigcap \mathcal{A}_i$ and $\bigcap \mathcal{D}_i$ are reducibility candidates. Since values of $\bigcap \mathcal{D}_i$ are values of data-candidates, $\bigcap \mathcal{D}_i \in \mathcal{DC}$.

(4.4) All candidates $\mathcal{A}_i$ satisfy (**CR3**), thus $\mathcal{A}_i = \overline{\mathcal{A}_i}$ for any i. So Lemma 4.14 says that $\overline{\bigcup \mathcal{A}_i} \subseteq \bigcup \mathcal{A}_i$. 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 (**CR1**), (**CR2**) and (**CR4**) are preserved by union. thus $\overline{\bigcup \mathcal{A}_i}$ is a reducibility candidate (by Lemma 4.7), and so is $\bigcup \mathcal{A}_i$.
In the same way, $\overline{\bigcup \mathcal{D}_i}$ is a reducibility candidate. By Remark 4.9, $\overline{\bigcup \mathcal{D}_i} \in \mathcal{DC}$.

(4.5) We prove that $S \to \mathcal{A}$ satisfy all conditions of (**CR**):

CR1. Let $t \in S \to \mathcal{A}$. There exists $u \in S$, and $tu \in \mathcal{A} \subseteq PN_0$. So $t \in PN_0$.

CR2. Let $t \in S \to \mathcal{A}$ and $t' \in Red_1(t)$. 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 $t' \in S \to \mathcal{A}$.

CR3. For any $t \in \mathcal{N}_D$ such that $Red_1(t) \subseteq S \to \mathcal{A}$, we prove that $u \in S$ implies $tu \in \mathcal{A}$ by induction on the reduction of u. Since $t \in \mathcal{N}_D$, tu is not a data-structure so $tu \in \mathcal{N}_D$. Furthermore t is not an abstraction so every reduct of tu is either $\maltese$ (if $t = \maltese$), or $t'u$ with $t' \in Red_1(t)$, or tu' with $u \to u'$. In any case it belongs to $\mathcal{A}$: $\maltese$ by (**CR3**), $t'u$ because $t' \in S \to \mathcal{A}$, and tu' by induction hypothesis. So $tu \in \mathcal{A}$ by (**CR3**), thus $t \in S \to \mathcal{A}$.

CR4. Let $t \to_c t'$ such that $t' \in S \to \mathcal{A}$. For any $u \in S$, $tu \to_c t'u$ and $t'u \in \mathcal{A}$. So $tu \in \mathcal{A}$ by (**CR4**) in $\mathcal{A}$.

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

(4.6) First notice that $\overline{\mathcal{D}\mathcal{A}} = \overline{\mathcal{D}\mathcal{A} \cup \maltese}$ (since $\maltese$ is neutral with no reduct, it is in the closure of any set). We call S the set $\mathcal{D}\mathcal{A} \cup \maltese$, and we will first prove that it is a non-expansed candidate. Then we will prove that $t' \in S$ and $t \to_c t'$ imply $t \in \overline{S}$. Also $\overline{S} \in \mathcal{CR}$ will result from Lemma 4.15.

– 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 $t \in PN_0$ and $Red_1(t) \subseteq S$. 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 $Red_1(t) \subseteq S$ and $t \in PN_0$. Hence S satisfy (**CR1**) and (**CR2**).

– Let $t \to_c t'$ 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 $t_i' \to_c t_i$ (in that case $t \in \mathcal{D}\mathcal{A}$ since $\mathcal{D}$ and $\mathcal{A}$ are closed by expansion for $\to_c$), or $t = \{\!|\theta|\!\} \cdot (t_0 t_2)$ and $t_1 = \{\!|\theta|\!\} \cdot t_0$. In the last case, $t \in \mathcal{N}_D$: 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 t may have three different forms:

- $u = t'$. Hence $u \in S \subseteq \overline{S}$.
- $u = \{\!|\theta|\!\} \cdot \maltese$ (if $t_0 = \maltese$). In that case $u \in \mathcal{N}_D$ and all its reducts in any number of steps until $\maltese$ are in $\mathcal{N}_D$, so u is in $\overline{S}$.
- $u = \{\!|\theta'|\!\} \cdot (t_0' t_2')$ with $\theta \to \theta'$ and $t_i = t_i'$, or $\theta = \theta'$ and $t_i \to t_i'$.
  In that case, $u \to_c u' = (\{\!|\theta'|\!\} \cdot t_0') t_2'$, and $t' \to u'$ so $u' \in S$ by (**CR2**). Thus $u \in \overline{S}$ by induction hypothesis.

Hence any reduct of t is in $\overline{S}$, and thus $t \in \overline{S}$ by (**CR3**).

By Lemma 4.15, $\overline{\mathcal{D}\mathcal{A}} = \overline{S} \in \mathcal{CR}$. What is more, all values of $\overline{\mathcal{D}\mathcal{A}}$ are in $\mathcal{D}\mathcal{A}$, thus they are applications, so they are data-structures. Finally, $\overline{\mathcal{D}\mathcal{A}} \in \mathcal{DC}$. □

In (4.6) 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 u to t if t is not (and does not reduce on) an abstraction.

Example 4.17. Consider the reducibility candidate $\mathcal{A} = \overline{\{I\}}$, where $I = \lambda x.x$
Then $II \in \overline{\mathcal{A}\mathcal{A}}$, but $II \to I$ and $I \notin \overline{\mathcal{A}\mathcal{A}}$. Thus $\overline{\mathcal{A}\mathcal{A}}$ is not closed under (**CR2**) and thereby is not a reducibility candidate.

5. Reducibility model

In this section 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.

5.1. **Modelling types.** 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. 6. We also associate to T (seen as a type for case bindings) and ρ the set of case bindings $[\![T]\!]_\rho$. Lemma 4.16 ensures that for every valuation ρ, $[T]_\rho \in \mathcal{CR}$ for any type T, and $[D]_\rho \in \mathcal{DC}$ for any data type D.

Type interpretation by reducibility candidates:

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

Interpretation of types for case bindings:

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

Figure 6: Interpretation of types

Note 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 (**CR3**): 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 $\{\!|c \mapsto tu|\!\} \cdot c$ is $tu \in \mathcal{DT}$, but $\{\!|c \mapsto tu|\!\} \cdot c$ 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 5.1. *If t is a value of $[\mathbf{c}T_1 \dots T_k]_\rho$ then $t = \mathsf{c}t_1 \dots t_k$ with $t_i \in [T_i]_\rho$.*

In particular, Proposition 1.1 ensures that $t \in [\mathbf{c}T_1 \dots T_k]_\rho$ implies $t \to^* \mathsf{c}t_1 \dots t_n$ for some $t_i \in [T_i]_\rho$ $(i \leq n)$, or $t \to^* \maltese$.

Proof. We proceed by induction on k.

If $k = 0$, it is straightforward from the definition of $[\mathbf{c}]_\rho$.

Else $[\mathbf{c}T_1 \dots T_k]_\rho = \overline{[\mathbf{c}T_1 \dots T_{k-1}]_\rho [T_k]_\rho}$, so

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

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 $u = \mathtt{c}t_1 \dots t_{k-1}$ with $t_i \in [T_i]_\rho$, and we conclude with $t_k = u' \in [T_k]_\rho$. □

Corollary 5.2. *For any constructor* $\mathtt{c}$ *and any types* $T_1, \dots, T_k$,

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

Proof. By Proposition 5.1, $\mathcal{V}al([\mathbf{c}T_1 \dots T_k]_\rho) = \mathbf{c}[T_1]_\rho \dots [T_k]_\rho$.
Since $\mathcal{V}al(\overline{c[T_1]_\rho \dots [T_k]_\rho})$ also is $\mathbf{c}[T_1]_\rho \dots [T_k]_\rho$, Corollary 4.12 entails the equality. □

The following lemma expresses that type interpretation is sound w.r.t. sub-typing.

Lemma 5.3. *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 straightforward from the definition. So are union and intersection rules. Introduction and elimination rules for quantifiers $\forall$ and $\exists$ use the equality $[T]_{\rho,\nu\mapsto[U]_\rho} = [T\{\nu \leftarrow U\}]_\rho$.

`Arrow` is standard, and `Constr` comes from Proposition 5.1: $[\mathbf{c}_1 \overrightarrow{T}] \cap [\mathbf{c}_2 \overrightarrow{U}]_\rho$ has no value if $\mathtt{c}_1 \neq \mathtt{c}_2$ and thus is smallest than any candidate.

We detail rules `App` and `Data`, other rules are easy to check (we actually introduced them in the calculus because they were valid in the model).

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

Remark that $\mathcal{D} \subseteq \mathcal{D}'$ and $\mathcal{T} \subseteq \mathcal{T}'$ implies $\mathcal{D}\mathcal{T} \subseteq \mathcal{D}'\mathcal{T}'$, and notice that the closure operator is monotone 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 $t \in [T \to DT]_\rho$. □

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

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:

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

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 $(x : T) \in \Gamma$

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 $\sigma \in [\Gamma]_\rho$,

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

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

Lemma 5.4. *For any $\mathcal{A} \in \mathcal{CR}$, any terms $t, u, \lambda x.t_0$, and every non-empty non-expansed candidate S,*

$$tu \in \mathcal{A} \quad \Leftrightarrow \quad t \in Red_*(u) \to \mathcal{A} \tag{5.1}$$

$$\lambda x.t_0 \in S \to \mathcal{A} \quad \Leftrightarrow \quad \text{for all } s \in S,\ t_0[x := s] \in \mathcal{A} \tag{5.2}$$

Proof.

(5.1) If $tu \in \mathcal{A}$ then for any $u' \in Red_*(u)$, $tu \to^* tu'$ hence $tu' \in \mathcal{A}$ by (**CR2′**). So $t \in Red_*(u) \to \mathcal{A}$. Conversely, if $t \in Red_*(u) \to \mathcal{A}$ then $tu \in \mathcal{A}$ since $u \in Red_*(u)$.

(5.2) If $\lambda x.t_0 \in S \to \mathcal{A}$, then for any $s \in S$, $(\lambda x.t_0)s \in \mathcal{A}$, so $(\lambda x.t_0)s \to t_0[x := s]$ implies $t_0[x := s] \in \mathcal{A}$ by (**CR2**). Now, if $t_0[x := s] \in \mathcal{A}$ for some $s \in S$, then $t_0 \in PN_0$ by Lemma 1.2. Moreover, For any $s' \in S$, we can easily check by induction on the reduction of t_0 and s' that $(\lambda x.t_0)s' \in \mathcal{A}$; indeed, it is in $\mathcal{N}_D$, and all its reducts are in $\mathcal{A}$. □

Remark 5.5. If $u \in PN_0$, then $Red_*(u)$ is a non-expansed candidate, and so $Red_*(u) \to \mathcal{A} \in \mathcal{CR}$ by (4.5). Also, if $u_i \in PN_0$ 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 (5.1) and an induction on k.

Lemma 5.6. *Let $\mathcal{A}_1, \dots, \mathcal{A}_k, \mathcal{B} \in \mathcal{CR}$ and $\theta \in PN_0$ Assume $\mathtt{c} \mapsto u \in \theta$, with $u \in \overrightarrow{\mathcal{A}} \to \mathcal{B}$ (where $\overrightarrow{\mathcal{A}} = \mathcal{A}_1; \dots; \mathcal{A}_k$). Then*

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

Proof. We prove that for all $\theta \in PN_0$ with $\mathtt{c} \mapsto u \in \theta$ and $u \in \overrightarrow{\mathcal{A}} \to \mathcal{B}$, and for all $t \in \overline{\mathtt{c}\mathcal{A}_1 \dots \mathcal{A}_k}$, the term $\{\!|\theta|\!\} \cdot t$ is in $\mathcal{B}$.

If t is a value then $t = \mathtt{c}t_1 \dots t_k$ with $t_i \in \mathcal{A}_i$, so

$\{\!|\theta|\!\} \cdot t \in \mathcal{B}$ iff $(\{\!|\theta|\!\} \cdot \mathtt{c})t_1 \dots t_k \in \mathcal{B}$ (**CR2′**), (**CR4′**)

iff $\{\!|\theta|\!\} \cdot \mathtt{c} \in Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$ (Remark 5.5)

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 $\{\!|\theta|\!\} \cdot \mathtt{c}$ is in $\overrightarrow{\mathcal{A}} \to \mathcal{B}$: this term is in $\mathcal{N}_D$ and its reducts are either $\{\!|\theta'|\!\} \cdot \mathtt{c}$ with $\theta \to \theta'$ (that is in $\overrightarrow{\mathcal{A}} \to \mathcal{B}$ by induction hypothesis), or u (that is in $\overrightarrow{\mathcal{A}} \to \mathcal{B}$ by hypothesis). So $\{\!|\theta|\!\} \cdot \mathtt{c}$ is in $\overrightarrow{\mathcal{A}} \to \mathcal{B}$ by (**CR3**), 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 $\{\!|\phi|\!\} \cdot h_0$ and $n \ge 0$, or $h = \lambda x.h_0$ and $n \ge 1$. We prove that $\{\!|\theta|\!\} \cdot t$ is in $\mathcal{B}$ by induction on the reductions of θ and h.

- First consider cases $h = \maltese$ or $\{\!|\phi|\!\} \cdot h_0$, and $n \ge 0$:

  $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$ iff $(\{\!|\theta|\!\} \cdot h)t_1 \dots t_k \in \mathcal{B}$ (**CR2′**), (**CR4′**)

  iff $\{\!|\theta|\!\} \cdot h \in Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$ (5.1)

  Note that $\{\!|\theta|\!\} \cdot h \in \mathcal{N}_D$ and $Red_*(t_1) \to \dots \to Red_*(t_k) \to \mathcal{B}$ is a reducibility candidate

by (4.5). 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 $\{\!|\theta'|\!\} \cdot h' \in Red_*(t_1) \to \cdots \to Red_*(t_k) \to \mathcal{B}$ by induction hypothesis. So $\{\!|\theta|\!\} \cdot h \in Red_*(t_1) \to \cdots \to Red_*(t_k) \to \mathcal{B}$ by (**CR3**), and $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$.

- Now consider case $h = \lambda x.h_0$ (with $x \notin \mathcal{FV}(\theta)$), and $n \geq 1$.
  $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$ iff $(\lambda x.\{\!|\theta|\!\} \cdot h_0)t_1 \dots t_k \in \mathcal{B}$ (**CR2′**), (**CR4′**)
  iff $\lambda x.\{\!|\theta|\!\} \cdot h_0 \in Red_*(t_1) \to \cdots \to Red_*(t_k) \to \mathcal{B}$ (5.1)
  iff *for all* $s \in Red_*(t_1)$,
  $\{\!|\theta|\!\} \cdot h_0[x := s] \in Red_*(t_2) \to \cdots \to Red_*(t_k) \to \mathcal{B}$ (5.2)
  For any $s \in Red_*(t_1)$, $t \to^* (\lambda x.h_0)\, s\, t_2 \dots t_n \to h_0[x := s]\, t_2 \dots t_n$, so that $\{\!|\theta|\!\} \cdot (h_0[x := s]t_2 \dots t_n) \in \mathcal{B}$ by induction hypothesis.
  Hence, $(\{\!|\theta|\!\} \cdot h_0[x := v])t_2 \dots t_n \in \mathcal{B}$ by (**CR2′**), and thus by (5.1), $\{\!|\theta|\!\} \cdot h_0[x := v]$ belongs to $Red_*(t_2) \to \cdots \to Red_*(t_k) \to \mathcal{B}$. Also $\{\!|\theta|\!\} \cdot t \in \mathcal{B}$.

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

Proposition 5.7. *Given a term t, a case binding θ, a context Γ and a type T,*

$$\Gamma \vdash t : T \quad \Rightarrow \quad \Gamma \vDash t : T \tag{5.3}$$

$$\Gamma \vdash \theta : T \quad \Rightarrow \quad \Gamma \vDash \theta : T \tag{5.4}$$

Proof. The proof is made 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** 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 (5.2).
If it comes from **Inter**, **Union**, or **Univ** it is straightforward from induction hypothesis. If it comes from **Subs**, it is a consequence of Lemma 5.3. We detail the proof in case the derivation comes from rule **CB** or **Exist** (**Inter** is similar to this last one).

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

Remember that the interpretation of a type T, seen as a type for case bindings is $[\![T]\!] = \{\theta \;/\; \lambda x.\{\!|\theta|\!\} \cdot x \in [T]\}$. Note $(U_{i1} \dots U_{ik}) = \overrightarrow{U_i}$, choose ρ a valuation and $\sigma \in [\Gamma]_\rho$, and show that $\lambda x.\{\!|\theta_\sigma|\!\} \cdot x \in [\mathbf{c}_i \overrightarrow{U}_i \to T_i]_\rho$. Let $t \in [\mathbf{c}_i \overrightarrow{U}_i]_\rho$. 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 Corollary 5.2, $t \in \overline{\mathbf{c}_i[U_{i1}]_\rho \dots [U_{ik}]_\rho}$, and $u_i \in [U_{i1}]_\rho \to \cdots \to [U_{ik}]_\rho \to [T_i]_\rho$ by induction hypothesis. All terms in θ_σ are perfectly normalising, so we can use Lemma 5.6 to get $\{\!|\theta_\sigma|\!\} \cdot t \in [T_i]_\rho$.

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

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

Remark 5.8. For a closed term t and a closed type T we immediately get

$$[T] \in \mathcal{CR} \qquad \text{and} \qquad \vdash t : T \;\Rightarrow\; t \in [T] \tag{5.5}$$

5.3. **Results from the model.** Remembering that reducibility candidates are included in PN_0, an immediate consequence of Remark 5.8 is the perfect normalisation of typed $\lambda^-_{\mathcal{C}}$-calculus.

Theorem 5.9. *Every well typed term is perfectly normalising for* $\lambda^-_{\mathcal{C}}$.

Furthermore, every closed and defined normal form is a value or the Daimon (Proposition 1.1). Since the Daimon is never created by a reduction step, typing a term ensures that it reduces strongly —and without case composition— on a value. We can even be more precise when using data types: if a term (written without ✠) has type $\mathbf{c}T_1 \dots T_k$, then it reduces on a data structure $\mathtt{c}t_1 \dots t_k$ (Proposition 5.1).

Now we call *pure value* a data structure whose all sub-terms are data structures (such as `cons 0 (cons (S(S0)) nil)` for instance) and *pure data type* a data type whose all sub-terms 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 **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 Proposition 5.1).

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

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

(where a pure value of $\mathbf{c}D_1 \dots D_k$ has form $\mathtt{c}v_1 \dots v_k$ 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

Typed lambda calculus with constructors provides a powerful polymorphic type system, with a notion of data types and type application. The difficulty of typing the commutation rule between case and application is overcome with a sub-typing system. In this paper we have shown that this type system ensures strong normalisation without match failure if *we remove the composition of case analysers* from the calculus. We can safely do so, since the case composition rule is not computationally necessary. However, we thus lose the separation property for the lambda calculus with constructors.

Related works. The first presentation of the pattern calculus [13] comes with a ML-style type system. This type system is less expressive than ours and does not prevent match failure during reduction, but it is decidable.

A more elaborated calculus, the *extension calculus*, was recently developed in [14]. It is typed with an extension of *System F à la Church*, that provides type application and also a pattern matching mechanism on types. A proof of strong normalisation, using the method based on reducibility candidates, is done for a restriction of this system. Although no type inference algorithm exists for this calculus, it has been implemented in `bondi` [7].

Several Church-style type systems have been proposed for the ρ-calculus, including a family of type systems organised in a cube similar to Barendregt's. As far as we know, no Curry-style type system has been proposed for the ρ-calculus.

Future works. This paper has raised many questions, mainly concerning a possible implementation of lambda calculus with constructors. The first one is about recursively defined data types, such as

$$\mathit{nat} \equiv \mathtt{0} \cup \mathtt{succ}(\mathit{nat}) \qquad ; \qquad \mathit{list}\ T \equiv \mathtt{nil}\ \cup\ \mathtt{cons}\ T\ (\mathit{list}\ T)$$

Adding a double sub-typing judgement for each data type is a way to do it, but it requires checking the correctness of each rule. A fixpoint operator would probably be a better way, since it would allow to add recursive data types "on the fly".

Still with the view to implementing $\lambda_{\mathcal{C}}$-calculus, we need to isolate a decidable fragment of our type system. This is a real challenge when it comes to type case bindings (remind the example of Section 2.3 page 7) and to use union types.

Last, it could be interesting to develop a denotational semantic for the lambda calculus with constructors. Since the literature about denotational semantics for pure lambda calculus (based on domain theory for instance) is abundant, we could try to adapt it to our calculus. An idea to do that, is to first traduce $\lambda_{\mathcal{C}}$-calculus into pure λ-calculus (in the spirit of CPS translations).

5.3.1. *Acknowledgements.* I started this work at the University of Buenos Aires, which hosted me for 6 months during my master thesis. I would like to thank Ariel Arbiser, Eduardo Bonelli, Carlos Lombardi, Alejandro Ríos and Roel de Vrijer for all the discussions we had there, and that were profitable for this paper. I also acknowledge my supervisor, Alexandre Miquel, for his helpful advice.

References

[1] The agda proof assistant. `http://wiki.portal.chalmers.se/agda/`.
[2] A. Arbiser, A. Miquel, and A. Ríos. A lambda-calculus with constructors. In *Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2006.
[3] A. Arbiser, A. Miquel, and A. Ríos. The lambda-calculus with constructors: Syntax, confluence and separation. *Journal of Functional Programming*, 19(5):581–631, 2009.
[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1984.
[5] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages*, pages 250–261, 2003.
[6] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*, volume 25 of *Texts in Theoretical Computer Science*. EATCS, 2004.
[7] Bondi, a programming language centred on pattern-matching. `http://www-staff.it.uts.edu.au/~cbj/bondi/`.
[8] H. Cirstea and C. Kirchner. Rho-calculus, its syntax and basic properties. In *5th International Workshop on Constraints in Computational Logics*, 1998.
[9] J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.
[10] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
[11] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice Hall, 1968.
[12] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the programming language Haskell, a non-strict, purely functional language (Version 1.2). Sigplan Notices, 1992.

[13] C. B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26(6):911–937, 2004.
[14] C. B. Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.
[15] C. B. Jay and D. Kesner. Pure pattern calculus. In *European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 2006.
[16] X. Leroy. The objective caml system. `http://caml.inria.fr/`.
[17] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
[18] J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.
[19] B. Petit. A polymorphic type system for the lambda-calculus with constructors. In *Typed Lambda Calculus and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 234–248, 2009.
[20] C. Riba. On the stability by union of reducibility candidates. In *Foundations of Software Science and Computation Structure*, volume 4423 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2007.