Encoding $\mathbb{C}$Duce in the $\mathbb{C}\pi$-Calculus*

Giuseppe Castagna[1], Mariangiola Dezani-Ciancaglini[2], and Daniele Varacca[3]

[1] École Normale Supérieure de Paris
[2] Universitá di Torino
[3] Imperial College London

Abstract. We present a type faithful encoding of $\mathbb{C}$Duce into the $\mathbb{C}\pi$-calculus. These calculi are two variants of, respectively, the λ-calculus and the π-calculus, characterised by rich typing and subtyping systems with union, negation, and intersection types.

The encoding is interesting because it sheds new light on the Milner-Turner encoding, on the relations between sequential and remote execution of functions/services, and on the validity of the equational laws for union and intersection types in π-calculus.

1 Introduction and Motivations

The language $\mathbb{C}$Duce [11,10] is a functional programming language for XML documents manipulation, with a very rich type system. Types and subtyping play a central role in $\mathbb{C}$Duce: for its design (patterns and pattern matching are built around types), for its execution (functions can be overloaded with run-time code selection), and for its implementation (pattern matching compilation and query computation use static type information to optimise execution). All these multifarious usages of types rely on a common foundational core: the *semantic subtyping* framework. An introduction to semantic subtyping can be found in [8], while [5] discusses several aspects and perspectives; technical details are given in [11,10]. In a nutshell, given a typed language with some (possibly recursive) type constructors (e.g., $\rightarrow$, $\times$, $\texttt{list}()$, ...), semantic subtyping is a technique to enrich the language with *type combinators*, i.e. set-theoretic union, intersection, and negation types. The behaviour of combinators is specified via the subtyping relation (rather than via the typing of the terms). The subtyping relation is "semantic" since instead of axiomatising it by a set of inference rules, one describes a set-theoretic interpretation of the types $[\![\ ]\!] : \mathbf{Types} \rightarrow \mathscr{P}(\mathscr{D})$ (where $\mathscr{P}$ denotes the powerset operator and $\mathscr{D}$ some domain) and then defines the subtyping relation as $s \leq t \stackrel{\text{def}}{\iff} [\![s]\!] \subseteq [\![t]\!]$. Such a set-theoretic interpretation must satisfy at least three design goals.

1. It must ensure that type *combinators* have a set-theoretic interpretation. This is done by imposing that union, intersection, and negation types are respectively interpreted as the set-theoretic union, intersection, and complement operations of $\mathscr{P}(\mathscr{D})$.
2. It must ensure that type *constructors* have a "natural" interpretation (at least, for what concerns subtyping), e.g., that product types are interpreted as set-theoretic products, function types as sets of maps from domain to co-domain, and so on.

* Work partially supported by FP6-2004-510996 Coordination Action TYPES, Cofin'04 project McTafi, Tralala ACI project, EPSRC grant GR/T04724/01 "Program Analysis and the Typed Pi-Calculus", and by an ENS visiting professorship grant for Mariangiola Dezani.

C. Baier and H. Hermanns (Eds.): CONCUR 2006, LNCS 4137, pp. 310–326, 2006.

3. It must allow for an interpretation of types as sets of values. This means that if we take as $\mathscr{D}$ the set of values of the language and as interpretation the function that maps a type to the set of all values of that type, then this new interpretation must induce the very same subtyping relation as the one used to type values.

Finding a domain $\mathscr{D}$ and an interpretation function $[\![-]\!]$ that satisfy the last two points is far from being trivial: a set-theoretic interpretation of functional and recursive types or the circularity between the typing of values and definition of subtyping are difficult constraints. As described in [8] and outlined later on, semantic subtyping provides a technique to do so.

In [6] the $\mathbb{C}\pi$-calculus is devised. $\mathbb{C}\pi$ is a type system for the π-calculus which exploits the same principles as $\mathbb{C}$Duce to enrich Pierce and Sangiorgi's types [16] with (set-theoretic) unions, intersections, and negations. In the cited paper, a higher-order extension of the $\mathbb{C}\pi$-calculus with functional values is discussed. However the question arises whether the extension is necessary or whether it is possible to encode functions as processes. It is well known that several such encodings are possible from the λ-calculus into the π-calculus [15,17,18]. In the Join-calculus language [9], the functional part is simply syntactic sugar for its coding in the concurrent part.

Contributions. In this paper we describe an encoding of $\mathbb{C}$Duce into the $\mathbb{C}\pi$-calculus. The encoding turned out not to be so straightforward as one may expect. The difficulty arises in finding an encoding of the types that respects the subtyping relation. The Milner-Turner translation of arrow types [17] respects the subtyping relation in the context of the simply typed λ-calculus, but it breaks down in the presence of intersection types.

Strictly speaking the technical contribution of this paper is twofold: first it introduces the local $\mathbb{C}\pi$-calculus, a variant of the $\mathbb{C}\pi$-calculus that admits unrestricted recursion on types, a feature not allowed in the version of the calculus presented in [6]; secondly it defines an encoding of $\mathbb{C}$Duce (hence, of intersection, union, and negation types) into local $\mathbb{C}\pi$ that preserves the typing and subtyping relations as well as the reduction semantics.

But beyond these technicalities, or actually hidden right in the technical details, there lies the main interest of this work. As we detail in Sections 4 and 5, the translation sheds new light on the Milner-Turner encoding as it shows the respective roles of argument and return channel that are used to simulate functions in a concurrent world. In particular, it shows that in the presence of type-case constructions, the latter must be scrambled by introducing some noise at the type level so that the receiver cannot gain information by testing the type of the return channel. The translation is a further confirmation of the validity of the equational laws for union and intersection types in the π-calculus, since a different axiomatisation proposed in the literature is incompatible with the Milner-Turner technique. This is not the only contribution to the type theory of the π-calculus, since the encoding also outlines the different roles played by the two contra-variant constructors of $\mathbb{C}\pi$, namely input channel and negation, and shows how they interplay when considering them from a logical point of view. Finally, at term level the translation formalises the nice correspondence between functional pattern matching and π-calculus guarded sums on a same input channel.

Structure. In Section 2 we present the local variant of the $\mathbb{C}\pi$-calculus. In Section 3 we present the functional core of $\mathbb{C}$Duce. Section 4 is devoted to explaining the main difficulties we encountered when encoding $\mathbb{C}$Duce types into $\mathbb{C}\pi$ types. Section 5 contains the formalisation of the encoding of the language, while Section 6 presents the correctness results. In Section 7 we conclude by giving some insight on more general aspects of this work and trying to convey the intuition of why we believe that the main contribution lies well beyond the technical result we present. For lack of space some definitions and all proofs are omitted; the interested reader can find them in [7].

2 The $\mathbb{C}\pi$-Calculus

The $\mathbb{C}\pi$-calculus is a variant of the asynchronous π-calculus with pattern matching in input and rich typing and subtyping systems [6]. We introduce here a further simplification of the calculus, following ideas of the Join calculus [9] and of the local π-calculus [14]. The key idea is that if a process is communicated a channel, then it cannot use that channel in input. Only global channels already known to the process or newly generated channels can be used in input. This policy is enforced syntactically, even before processes are typed. In the typing system, this implies that input channel types are no longer necessary. The consequent subtyping relation is much easier to decide and, unlike the system for full $\mathbb{C}\pi$-calculus, can be also extended to recursive types.

2.1 Types and Subtyping

A type is coinductively defined by applying *type constructors*, namely base type constructors (e.g. integers, strings, etc...), the channel or product type constructors, or by applying a *boolean combinator*, i.e., union, intersection, and negation. More formally, types are regular trees generated by the following grammar

$$\begin{array}{lrcll} \mathbb{C}\pi\ \textit{Types} & t & ::= & \mathbf{b} \mid \overline{ch}(t) \mid t \times t & \text{constructors} \\ & & \mid & \mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t & \text{combinators} \end{array}$$

and that are contractive, that is for which on every infinite branch of the tree there are infinitely many occurrences of constructors. Combinators are self-explaining, with $\mathbf{0}$ being the empty type and $\mathbf{1}$ the type of all values. We use $\mathbf{b}$ to range over base types. The channel type constructor $\overline{ch}(t)$ denotes the type of channels that can be used to *output* values of type t. The set of all types (sometimes referred to as "type algebra") will be denoted by $\mathscr{T}$. Contractivity ensures, as usual, the absence of meaningless recursively defined types such as $t = \neg t$. The subtyping relation is defined semantically. This means that we first give a set-theoretical interpretation of types $[\![-]\!] : \mathscr{T} \to \mathscr{P}(\mathscr{D})$, for some domain $\mathscr{D}$, and then define subtyping as inclusion of the interpretations: $s \leq t \stackrel{\text{def}}{\iff} [\![s]\!] \subseteq [\![t]\!]$. It is out of the scope of this work to precisely define $\mathscr{D}$ or the interpretation $[\![-]\!]$ (see [6] for details). All we need for this work is to precisely define the subtyping relation these definitions entail. This is *completely* characterised by the subtyping relation on the basic types and by the following property:

$$[\![s]\!] \subseteq [\![t]\!] \iff \mathbb{E}(s) \subseteq \mathbb{E}(t) \tag{1}$$

where $\mathbb{E}(-)$ is defined as follows

Definition 2.1 (Extensional interpretation). *The* extensional interpretation *of the types is the function* $\mathbb{E}(-) : \mathcal{T} \to \mathscr{P}(\mathscr{D} + \mathscr{D} \times \mathscr{D} + \mathscr{P}(\mathscr{D}))$*, defined as follows:*

a. $\mathbb{E}(\mathbf{1}) = \mathscr{D} + \mathscr{D} \times \mathscr{D} + \mathscr{P}(\mathscr{D})$, $\mathbb{E}(\mathbf{0}) = \varnothing$, $\mathbb{E}(\mathbf{b}) = [\![\mathbf{b}]\!]$;
b. $\mathbb{E}(t_1 \vee t_2) = \mathbb{E}(t_1) \cup \mathbb{E}(t_2)$, $\mathbb{E}(t_1 \wedge t_2) = \mathbb{E}(t_1) \cap \mathbb{E}(t_2)$, $\mathbb{E}(\neg t) = \mathbb{E}(\mathbf{1}) \setminus \mathbb{E}(t)$;
c. $\mathbb{E}(t_1 \times t_2) = [\![t_1]\!] \times [\![t_2]\!]$;
d. $\mathbb{E}(ch\bar{}(t)) = \{[\![s]\!] \mid [\![s]\!] \supseteq [\![t]\!]\}$.

The intuition underlying property (1) is that, *for what concerns subtyping*, we can consider that $[\![t]\!]$, i.e. the semantics of t, is precisely $\mathbb{E}(t)$. Thus in Definition 2.1, (b.) states that the type combinators are interpreted as the corresponding set operations and (c.) that the product type is interpreted as set-theoretic product. Point (d.) gives us the semantics of channels. Intuitively, if a type denotes the set of all values with that type, then the type $ch\bar{}(t)$ denotes the set of all channels in which one can safely put objects of type t. Therefore it will denote all channels that can contain objects of type s, for any $s \geq t$. Let us write c^t for a channel named c and transporting objects of type t. We have $[\![ch\bar{}(t)]\!] = \{c^s \mid s \geq t\}$. The derived subtyping relation is insensitive to the actual number of channels of a given type or to their names. We can therefore assume that for every equivalence class of types, there is only one such channel, which may as well be identified with $[\![t]\!]$, so that the intended semantics of channel types would be

$$[\![ch\bar{}(t)]\!] = \left\{ [\![s]\!] \mid s \geq t \right\} \tag{2}$$

which by definition of subtyping gives point (d.) of the previous definition. The subtyping relation of the local ℂπ-calculus is decidable and the decision algorithm is much simpler than the one for the full ℂπ-calculus presented in [6].

In order to stress that property (1) and Definition 2.1 completely define the subtyping relation, let us show as an example how to deduce the contra-variance of the output type channel constructor: $ch\bar{}(s) \leq ch\bar{}(t) \Leftrightarrow [\![ch\bar{}(s)]\!] \subseteq [\![ch\bar{}(t)]\!] \Leftrightarrow \mathbb{E}(ch\bar{}(s)) \subseteq \mathbb{E}(ch\bar{}(t)) \Leftrightarrow \{[\![u]\!] \mid [\![u]\!] \supseteq [\![s]\!]\} \subseteq \{[\![u]\!] \mid [\![u]\!] \supseteq [\![t]\!]\} \Leftrightarrow [\![t]\!] \subseteq [\![s]\!] \Leftrightarrow t \leq s$.

Similarly we can derive interesting equations and inequations between types. For instance, $ch\bar{}(t) \leq ch\bar{}(\mathbf{0})$ is a special case of the contra-variance we just derived. It states that every channel c can be safely used in a process that does not write on c. If we define $s = t \overset{\text{def}}{\Longleftrightarrow} [\![s]\!] = [\![t]\!]$, then we have

$$ch\bar{}(t_1) \wedge ch\bar{}(t_2) = ch\bar{}(t_1 \vee t_2) \tag{3}$$

which states that if on a channel we can write values of type t_1 and values of type t_2, then we can also write values of type $t_1 \vee t_2$, and vice versa. Union of channel types behaves differently since the inequation below is strict (see [6] for a discussion on this)

$$ch\bar{}(t_1) \vee ch\bar{}(t_2) \;\lneq\; ch\bar{}(t_1 \wedge t_2)\ .$$

2.2 Patterns

Both ℂπ and ℂDuce feature powerful pattern matching. Patterns perform type-cases, decompose values by capturing subcomponents in variables, and can be recursive.

Definition 2.2. *Given a type algebra $\mathscr{T}$, and a set of variables $\mathbb{V}$, a* pattern *p on $(\mathbb{V},\mathscr{T})$ is a regular tree generated by the following grammar*

$$p ::= x \mid t \mid p \wedge p \mid p|p \mid (p,p)$$

such that (i) on every infinite branch of p there are infinitely many occurrences of the pair pattern, (ii) for every subterm $p_1 \wedge p_2$ of p we have $Var(p_1) \cap Var(p_2) = \varnothing$, and (iii) for every subterm $p_1|p_2$ of p we have $Var(p_1) = Var(p_2)$ (where $x \in \mathbb{V}$, $t \in \mathscr{T}$, and $Var(p)$ is the set of variables occurring in p).

The semantics of patterns is given in terms of a matching operation that returns either a substitution for the variables of the pattern or a failure denoted by Ω. Matching can be defined independently from the language, via the domain $\mathscr{D}$ of a model of types. We use d/p to denote the matching of the element d against the pattern p. Intuitively, x is the pattern that always succeeds and captures the matched element in x (i.e., d/x returns the substitution $\{x \mapsto d\}$); t succeeds if the element is in the interpretation of t, in which case it returns the empty substitution; the intersection succeeds only if both patterns succeed and it returns the union of the substitutions; the alternation follows a first-match policy by applying the pattern on the right only if the one on the left failed; the pair decomposes the element and applies the patterns to the respective sub-components. See [6] for the formal definition. It can be shown that the set of all elements for which a pattern p does not fail is the denotation of a type. We denote this type by $\lfloor p \rfloor$, that is by definition $[\![\lfloor p \rfloor]\!] = \{d \mid d/p \neq \Omega\}$. Matching can be extended to types as stated by the following theorem:

Theorem 2.3 (A.5 in [11]). *There is an algorithm mapping every pair (t,p), where p is a pattern and t a type such that $t \leq \lfloor p \rfloor$, to a type environment $(t/p) \in \mathscr{T}^{Var(p)}$ such that $[\![(t/p)(x)]\!] = \{(d/p)(x) \mid d \in [\![t]\!]\}$.*

2.3 The Language

The syntax of $\mathbb{C}\pi$ is similar to that of the asynchronous π-calculus [3,13], extended with call-by-value pattern matching (obtained by pattern-guarded sums of inputs on the same channel) and an extra condition which guarantees "locality" [14].

Processes $P ::=$		
	$\overline{\alpha}M$	output
$\mid$	$\sum_{i\in I} c^t(p_i).P_i$	patterned input
$\mid$	$P \parallel P$	parallel
$\mid$	$(\nu c^t)P$	restriction
$\mid$	$!P$	replication

Channels $\alpha ::=$		
	x	variables
$\mid$	c^t	constant

Messages $M ::=$		
	n	constants
$\mid$	α	channel
$\mid$	(M,M)	pair

where I is a possibly empty finite set of indexes, t ranges over the types defined in Section 2.1 and p_i are patterns as defined in Definition 2.2. As customary, empty sum corresponds to the inert process, denoted by 0. The *values* of the language are the closed messages $v ::= n \mid c^t \mid (v,v)$. We use $\mathscr{V}$ to denote the set of all values.

Observe that we force input to happen on channel constants. This ensures that channels sent by other processes cannot be used in input. Output instead can be performed on non-constant channels, too. Since pattern matching performs type-case, we must define the typing of messages before the reduction semantics, see Figure 1. We suppose

Messages

$$\frac{}{\Gamma \vdash n : \mathbf{b}_n}\ (\text{const}) \qquad \frac{s_i \not\leq t}{\Gamma \vdash c^t : ch(t) \wedge \neg ch(s_1) \wedge \ldots \wedge \neg ch(s_n)}\ (\text{chan})$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{var}) \qquad \frac{\Gamma \vdash M : s \leq t}{\Gamma \vdash M : t}\ (\text{subs}) \qquad \frac{\Gamma \vdash M_1 : t_1,\ \Gamma \vdash M_2 : t_2}{\Gamma \vdash (M_1, M_2) : t_1 \times t_2}\ (\text{pair})$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ (\text{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{repl}) \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ (\text{para})$$

$$\frac{t \leq \bigvee_{i \in I} \lfloor p_i \rfloor \quad \Gamma, (t \wedge \lfloor p_i \rfloor)/p_i \vdash P_i}{\Gamma \vdash \sum_{i \in I} c^t(p_i).P_i}\ (\text{input}) \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : ch(t)}{\Gamma \vdash \overline{\alpha} M}\ (\text{output})$$

Fig. 1. ℂπ typing rules

that every basic constant n is associated to an atomic basic type $\mathbf{b}_n$. The rules, and in particular rule (chan), are designed so that we can interpret a type as the set of all values of that type. The interpretation $[\![\ ]\!]_{\mathscr{V}} : \mathscr{T} \to \mathscr{P}(\mathscr{V})$ defined as

$$[\![t]\!]_{\mathscr{V}} = \{v \mid \varnothing \vdash v : t\} \tag{4}$$

satisfies property (1) and, furthermore, it generates the same subtyping relation as $\leq$.[1]

Then, the definition for pattern matching given in Section 2.2 applies for v being a value and we can use it to define the reduction semantics of ℂπ:

$$\overline{c^t} v \ \| \ \sum_{i \in I} c^t(p_i).P_i \quad \longrightarrow \quad P_j[v/p_j]$$

where $P[\mathbf{s}]$ denotes the application of substitution $\mathbf{s}$ to process P. The asynchronous output of a *value* on the channel c^t synchronises with a summand in a sum guarded by the same channel, only if the pattern of the summand matches the communicated value (the type system ensures the existence of such a pattern). If more than one pattern matches, then one of them is non-deterministically chosen and the corresponding process executed, but before its execution the pattern variables are replaced by the captured values. As usual, the notion of reduction must be completed with reductions in evaluation contexts and up to structural congruence, whose definitions are standard and can be found in [6]. We use $\longrightarrow^*$ to denote the reflexive and transitive closure of $\longrightarrow$.

The typing of processes is defined in the lower half of Figure 1. Notice that the rule for restrictions (new) does not rely on the type environment Γ, since channels are decorated by the type of their messages, and that in the rule (input) the condition $t \leq \bigvee_{i \in I} \lfloor p_i \rfloor$ ensures that for every message that may arrive on the channel, there exists at least one pattern that matches it. The system satisfies subject reduction [6].

[1] Without the intersection of the negated channel types in (chan), we could not prove that, say, $c^{\mathtt{int}} : \neg ch(\mathtt{bool})$. More generally, the property $\vdash v : t \ \Leftrightarrow \not\vdash v : \neg t$ would not hold, and this is necessary to $[\![-]\!]_{\mathscr{V}}$ to satisfy (1): cf. Definition 2.1(b). For a broader discussion on such inference rules with negated types see Section 4.6 of [5].

3 The Functional Language $\mathbb{C}$Duce

$\mathbb{C}$Duce is a very efficient functional language for rapid design and development of applications that manipulate XML data [2]. In this work we concentrate on the foundational aspects of $\mathbb{C}$Duce [11] a detailed survey of which can be found in [8]. In that respect, $\mathbb{C}$Duce features the same syntactic types as $\mathbb{C}\pi$, with just a single exception, namely, the channel type constructor is replaced by the function type constructor:

$$\begin{array}{lrll} \mathbb{C}\text{Duce } \textit{Types} & \tau ::= & \mathbf{b} \mid \tau \to \tau \mid \tau \times \tau & \text{constructors} \\ & \mid & \mathbf{0} \mid \mathbf{1} \mid \neg\tau \mid \tau \vee \tau \mid \tau \wedge \tau & \text{combinators} \end{array}$$

where the same regularity and contractivity restrictions as in Section 2.1 apply. We use σ, τ to range over $\mathbb{C}$Duce types and to typographically distinguish them from $\mathbb{C}\pi$ ones, these latter still ranged over by s and t. Subtyping is characterised in the same way as for $\mathbb{C}\pi$, by defining an interpretation from the above types into a domain $\mathscr{D}$ (that we leave unspecified, see [11]) which satisfies property (1). Definition 2.1 is modified to account for the new type constructor for functions. We have $\mathbb{E}(-) : \mathscr{T} \to \mathscr{P}(\mathscr{D} + \mathscr{D}\times\mathscr{D} + \mathscr{P}(\mathscr{D}\times\mathscr{D}_\Omega))$ (where $\mathscr{D}_\Omega = \mathscr{D} + \{\Omega\}$, the disjoint union of the domain and of a distinguished error element Ω) while point (d.) of Definition 2.1 becomes:

$$\text{d.}\quad \mathbb{E}(\sigma \to \tau) = \mathscr{P}\left(\overline{[\![\sigma]\!] \times \overline{[\![\tau]\!]}^{\mathscr{D}_\Omega}}^{\mathscr{D}\times\mathscr{D}_\Omega}\right)$$

where $\overline{X}^Y$ denotes the complement of X with respect to Y (i.e., $Y \setminus X$). In words, the extensional interpretation of $\sigma \to \tau$ is the set of graphs such that if the first element is in $[\![\sigma]\!]$, then the second element is in $[\![\tau]\!]$ (otherwise the second element can be anything, in particular the error Ω). Therefore, *for what concerns subtyping*, we can consider that arrow types are interpreted as follows:

$$[\![\sigma \to \tau]\!] = \{f \subseteq \mathscr{D} \times \mathscr{D}_\Omega \mid \forall (d_{\text{in}}, d_{\text{out}}) \in f.\ d_{\text{in}} \in [\![\sigma]\!] \Rightarrow d_{\text{out}} \in [\![\tau]\!]\}.$$

As we did for $\mathbb{C}\pi$, we can use this characterisation to deduce several type equality and containment relations.[2] For the goals of this work an utmostly interesting equation is

$$(\sigma \to \tau) \wedge (\sigma \to \tau') = \sigma \to \tau \wedge \tau' \tag{5}$$

whose validity can be easily checked by the reader, by applying the definition of $\mathbb{E}(-)$.

$\mathbb{C}$Duce is a λ-calculus with pairs, overloaded recursive functions, and pattern matching. This is reflected by the following syntax:

$$e ::= x \mid n \mid ee \mid (e,e) \mid \mu f^{\bigwedge_{i\in I}(\sigma_i\to\tau_i)}(x).e \mid \texttt{match}\ e\ \texttt{with}\ p{\Rightarrow}e \,|\, p{\Rightarrow}e$$

where patterns p are those defined in Definition 2.2 (but use $\mathbb{C}$Duce types). The typecase expression $(x = e \in \tau)\,?e_1 : e_2$ can be added as syntactic sugar for the matching expression $\texttt{match}\ e\ \texttt{with}\ x \wedge \tau {\Rightarrow} e_1 \,|\, x \wedge \neg\tau {\Rightarrow} e_2$.

[2] The error Ω is included in the codomain of the functions since without it every function would have type $\mathbf{1} \to \mathbf{1}$, therefore every application would be well-typed (with type $\mathbf{1}$). The error element Ω stands for the result of ill-typed applications. Thanks to it $\sigma \to \tau \leq \mathbf{1} \to \mathbf{1}$ does not hold in general, hence, it explicitly avoids the problem above.

$$\frac{}{\Delta;\Gamma \vdash n : \mathbf{b}_n}\,(const) \qquad \frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}\,(var) \qquad \frac{}{\Delta;\Gamma \vdash f : \Delta(f)}\,(fvar) \qquad \frac{\Delta;\Gamma \vdash e : \sigma \leq \tau}{\Delta;\Gamma \vdash e : \tau}\,(subs)$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash (e_1,e_2) : \tau_1 \times \tau_2}\,(pair) \qquad \frac{\Delta;\Gamma \vdash e_1 : \sigma \to \tau \quad \Delta;\Gamma \vdash e_2 : \sigma}{\Delta;\Gamma \vdash e_1 e_2 : \tau}\,(appl)$$

$$\frac{\begin{array}{l}(\text{for } \sigma_1 \equiv \sigma \wedge \lfloor p_1 \rfloor,\ \sigma_2 \equiv \sigma \wedge \neg \lfloor p_1 \rfloor) \\ \quad \Delta;\Gamma \vdash e : \sigma \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor \quad \Delta;\Gamma,(\sigma_i/p_i) \vdash e_i : \tau_i\end{array}}{\Delta;\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 : \bigvee_{\{i \mid \sigma_i \not\simeq \mathbf{0}\}} \tau_i}\,(match)$$

$$\frac{\begin{array}{l}(\text{for } \tau \equiv \bigwedge_{i\in I}(\sigma_i \to \tau_i)) \quad (\forall i \in I, h \in I, j \in J) \\ \quad \sigma_h \wedge \sigma_i = \mathbf{0} \qquad \tau \not\leq \sigma'_j \to \tau'_j \qquad \Delta, f : \tau; \Gamma, x : \sigma_i \vdash e : \tau_i\end{array}}{\Delta;\Gamma \vdash \mu f^{\tau}(x).e : \tau \wedge \bigwedge_{j\in J} \neg(\sigma'_j \to \tau'_j)}\,(abstr)$$

Fig. 2. ℂDuce typing rules

Function abstractions use a μ-abstracted name for recursion and specify at their index several arrow types, indicating that the function has all these types (i.e., their intersection). This is formally stated by the rule (*abstr*) in Figure 2 which for each $i \in I$ checks that the body e has type τ_i under the hypothesis that x has type σ_i. Note that the types of μ-abstracted variables are recorded in a distinct environment Δ. The distinction here is totally useless (we could have used a unique Γ) but it will be handy when we define the encoding (since μ-abstracted variables are translated into channel constants, then the encoding will be parametric only in Γ).

The only difficult rule is (*match*). It first deduces the type σ of the matched expression and checks whether patterns cover all its possible results (i.e., $\sigma \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor$); then it separately checks the first branch under the hypothesis that p_1 is selected (i.e. e is in $\sigma \wedge \lfloor p_1 \rfloor$) and the second branch under the hypothesis that p_2 is selected (i.e., e in $\sigma \wedge \neg \lfloor p_1 \rfloor$); finally it discards the return types of the branches *that cannot be selected*, which is safely approximated by the fact that the corresponding σ_i is empty.[3]

The rules in Figure 2 are the same as those defined in [11] (to which the reader can refer for more details) with just a single exception: in rule (*abstr*) we require that the arrows specified at the index of the function have disjoint domains: $\forall i, h < i . \sigma_h \wedge \sigma_i = \mathbf{0}$. This restriction is necessary (but not sufficient) in order to avoid the problem of output-driven overloading explained in Section 4.2. However, it causes no loss of generality, since every ℂDuce function $\mu f^{\bigwedge_{i\in I}(\sigma_i \to \tau_i)}(x).e$ can be put into this form by iterating on its index the rewriting that replaces $(\sigma_h \wedge \sigma_k \to \tau_h \wedge \tau_k) \wedge (\sigma_k \wedge \neg\sigma_h \to \tau_k) \wedge (\sigma_h \wedge \neg\sigma_k \to$

[3] The reader may wonder why the system does not return a type error when one of the two branches cannot be selected. As a matter of fact this is a key feature for typing overloaded functions, where the body is repeatedly checked under different hypothesis for some of which the σ_i of some typecase may be empty. This simple function should clarify the point: $\mu f^{(\mathsf{Int}\to\mathsf{Int};\mathsf{Bool}\to\mathsf{Bool})}(x).(y = x \in \mathsf{Int})\,?(y+1) : \mathsf{not}(y)$ when we type the body under the hypothesis $x : \mathsf{Int}$, then the second branch cannot be selected, while under $x : \mathsf{Bool}$ is the first one that cannot be selected. Without the selective union in the typing rule the best type we could have given to this function would have been $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$.

τ_h) for every pair of arrows $\sigma_h \to \tau_h$, $\sigma_k \to \tau_k$ such that $\sigma_h \wedge \sigma_k \neq \mathbf{0}$. This rewriting is sound and it is easy to show that the two functions are operationally indistinguishable (e.g., by applicative bisimilarity).

As the intersection of negated channels in the rule (chan) ensures that values of $\mathbb{C}\pi$ yield a model that induces the same subtyping relation as the initial one, so does for $\mathbb{C}$Duce the intersection of negated arrows in the rule (*abstr*): the interpretation defined by (4) where values are closed terms generated by $v ::= n \mid \mu f^{\bigwedge_{i\in I}(\sigma_i\to\tau_i)}(x).e \mid (v,v)$ and types are $\mathbb{C}$Duce types, enjoys the same properties. Therefore, we can again use the pattern semantics of Section 2.2 to define the call-by-value operational semantics of $\mathbb{C}$Duce (we omit the straightforward context rules that can be found in [11]).

$$\begin{array}{rcll} v_1 v_2 & \longrightarrow & e[v_1/f; v_2/x] & \text{if } v_1 = \mu f^{\tau}(x).e \\ \texttt{match}\ v\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 & \longrightarrow & e_1[v/p_1] & \text{if } v/p_1 \neq \Omega \\ \texttt{match}\ v\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 & \longrightarrow & e_2[v/p_2] & \text{if } v/p_1 = \Omega, v/p_2 \neq \Omega \end{array}$$

The calculus satisfies the subject reduction property [2].

4 Roadmap to the Encoding

In this section we discuss the main difficulties encountered in the definition of an encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$. It lists some failed attempts which will clarify the reasons behind the successful attempt.

4.1 The Milner-Turner Encoding

Since our encoding involves languages with subtyping, the first approach we tried was to adapt the Milner-Turner (MT) encoding of the call-by-value typed λ-calculus with subtyping into the typed π-calculus with subtyping, as presented in [17]. The translation of arrow types presented there is:

$$(\!|\sigma \to \tau|\!) \;=\; c\overline{h}((\!|\sigma|\!) \times c\overline{h}((\!|\tau|\!)))\ .$$

The encoding of λ-terms, decorated by their minimum types, is:

$$\begin{array}{ll} (\!|x^{\tau}|\!)_c^{\Gamma,x:\tau} & = \overline{c}(x) \\ (\!|\lambda x^{\sigma}.e^{\tau}|\!)_c^{\Gamma} & = (\nu\, a^{(\!|\sigma|\!)\times c\overline{h}((\!|\tau|\!))})(\overline{c}(a) \parallel\, !(a(x,b).(\!|e^{\tau}|\!)_b^{\Gamma,x:\sigma})) \\ (\!|e_1^{\sigma\to\tau} e_2^{\rho}|\!)_c^{\Gamma} & = (\nu\, a^{(\!|\sigma|\!)\times c\overline{h}((\!|\tau|\!))})(\nu\, b^{(\!|\rho|\!)})((\!|e_1^{\sigma\to\tau}|\!)_a^{\Gamma} \parallel\, a(w).((\!|e_2^{\rho}|\!)_b^{\Gamma} \parallel\, b(h).\overline{w}(h,c))) \end{array}$$

The encoding of an expression e is parametrised by a type environment Γ such that $\Gamma \vdash e : \tau$ and by a channel $c^{(\!|\tau|\!)}$ on which the value of the expression is returned to the environment. A function is represented by a channel (the "name" of the function) which can be called by sending the input value and a channel on which the output value should be returned. These two parameters are used by a replicated process (the "body" of the function) which returns the output value upon termination. In the encoding of the application, the encoding of the function is called on the encoding of the argument, and the returned value is returned as the value of the whole expression. This encoding bears a strong resemblance with the continuation passing style transform. In this sense, the return channel of an expression could be seen as the address of the continuation.

Since we translate only well-typed terms, in the case of the application we must have $\rho \leq \sigma$. The encoding of the application (in particular, the $\overline{w}(h,c)$ subterm) is well-typed only if this implies $(\!|\rho|\!) \leq (\!|\sigma|\!)$. This holds true in the simply typed λ-calculus with subtyping, but fails as soon as we add intersection types. In that case, the translation of the types does not preserve the identity of types: in ℂDuce, we have seen that the identity (5) holds (i.e., $(\sigma \to \tau) \wedge (\sigma \to \tau') = \sigma \to \tau \wedge \tau'$), while the same does not hold on the encodings of the types at issue since, in general, it is not true that

$$\overline{ch}(s \times \overline{ch}(t)) \wedge \overline{ch}(s \times \overline{ch}(t')) \leq \overline{ch}(s \times \overline{ch}(t \wedge t')) \ .$$

Using this observation we can indeed show that the MT encoding maps a well-typed ℂDuce expression into an ill-typed ℂπ process.

4.2 Output-Driven Overloading

In order to give an operational intuition of why the MT encoding does not work, recall that intersections of arrow types are commonly assimilated to the types of overloaded functions. In ℂDuce, the identity $(\sigma \to \tau) \wedge (\sigma \to \tau') = \sigma \to \tau \wedge \tau'$ is justified because overloaded functions can perform a type-case only on the type of the input. Therefore, if on the same input a function returns values of type τ *and* values of type τ' it must return only values that have both types.

In ℂπ, however, a process that encodes a function receives in input also the return channel. In principle such process could perform a type-case on this extra piece of information and then execute different computations according to whether the expected result is of type τ or τ'. Such "output-driven" overloaded function can, on the same input, return a value of type τ and a *different* value of type τ' (and not in τ). This is a function that is in $(\!|(\sigma \to \tau) \wedge (\sigma \to \tau')|\!)$ and not in $(\!|\sigma \to \tau \wedge \tau'|\!)$, therefore we expect that $(\!|\sigma \to \tau \wedge \tau'|\!) \lneq (\!|(\sigma \to \tau) \wedge (\sigma \to \tau')|\!)$ which is indeed the case.

4.3 The Distributive Law

At a first analysis, it may seem that the problem is the subtyping relation of ℂπ. We may be tempted to change it by adding the following inequation:

$$\overline{ch}(t_1 \wedge t_2) \ \leq \ \overline{ch}(t_1) \vee \overline{ch}(t_2) \ .$$

Since the converse inequality already holds (as seen in Section 2), we would obtain a "contravariant" distributive law of the channel constructor over the intersection. A similar distributive law is used by Hennessy and Riely in [12] to *define* the intersection type. As explained in [6], the above inequation is not justified in a calculus endowed with dynamic type-case. It is also not clear at first sight whether introducing the inequation is at all possible using a semantic approach. In any case, this new subtyping relation would not make the translation work either as it would introduce *too many* equations in the translation. For example, being $\mathtt{int} \wedge \mathtt{bool} = \mathbf{0}$, we would get

$$\overline{ch}(\mathbf{0} \times \overline{ch}(\mathtt{int} \vee \mathtt{bool})) \ \leq \ \overline{ch}(\mathtt{int} \times \overline{ch}(\mathtt{bool})) \vee \overline{ch}(\mathtt{bool} \times \overline{ch}(\mathtt{int})).$$

The type on the left is the encoding of $\mathbf{0} \to \mathtt{int} \vee \mathtt{bool}$ and the other type is the encoding of $(\mathtt{int} \to \mathtt{bool}) \vee (\mathtt{bool} \to \mathtt{int})$. This subtyping gives a problem already for the identity function, which has type $\mathbf{0} \to \mathtt{int} \vee \mathtt{bool}$ but not $(\mathtt{int} \to \mathtt{bool}) \vee (\mathtt{bool} \to \mathtt{int})$.

4.4 The Negation Translation

Intuitively, to find an encoding that respects type equality, we need that, when encoding the arrow type, the operator that encodes the output type distributes over the intersection, while the operator that encodes the input type should not distribute over the intersection. One possible encoding that satisfies this requirement is the following:

$$(\!|\sigma \to \tau|\!) \;=\; \overline{ch}(\neg(\!|\sigma|\!) \times (\!|\tau|\!)) \;.$$

Indeed the negation is a contravariant constructor that distributes over the intersection. However it was not clear to us what operational interpretation we could attach to this translation. Under this translation of the types, the MT translation of the λ-terms would not be well-typed.

This however was the sparkle that brought us to our solution: (i) We want to preserve the naturalness of the MT encoding, that is, to encode functions calls by RPCs that send along with the argument a channel on which the call must return the result; thus the type of the second argument of the call (i.e., the one that encodes the output type τ) must allow for messages of type $\overline{ch}((\!|\tau|\!))$. (ii) We also want the type of this argument to distribute over intersections, in order to respect the subtyping relation; the use of negation, $\neg(\!|\tau|\!)$, seems to help in this direction. Finally, (iii) we want this second argument to be contravariant (since it is under a $\overline{ch}(\,)$, it will then respect the covariance of the output type it is meant to encode); but the joint use of two contravariant constructors, $\overline{ch}(\,)$ and $\neg$, would make it covariant, thus we may need to add a further negation to make it contravariant. All this yields, for the encoding of $\sigma \to \tau$, a second argument of type $\neg(\overline{ch}(\neg(\!|\tau|\!)))$, which is *almost* what we are looking for. We say "almost" since it still does not satisfy (i) insofar as it is not a supertype of $\overline{ch}((\!|\tau|\!))$; as we will explain in Section 5.2 one point is still missing from it: $\overline{ch}(\mathbf{1})$ — to verify it, simply compute the difference $\overline{ch}((\!|\tau|\!)) \setminus \neg\overline{ch}(\neg(\!|\tau|\!))$. So we add it, obtaining for the second argument the following encoding $\neg\overline{ch}(\neg(\!|\tau|\!)) \vee \overline{ch}(\mathbf{1})$. This idea is carried out in details and generalised in the following section.

5 The Encoding

We propose a modification of the Milner-Turner encoding that respects type equality, and it is very close to the original translation.

5.1 The λ-Channel Constructor

The encoding of the types we propose is parametric with respect to a constructor of $\mathbb{C}\pi$ types that we call "λ-channel" type. This notion is designed to make the translation of types to respect the type equality (unlike the Milner-Turner and distributive approach), and to make the translation of terms to make sense (unlike the negation approach).

Definition 5.1. *A* λ-channel *(noted,* $ch^{\lambda}(-)$*) is a unary constructor of* $\mathbb{C}\pi$ *types s.t.:* (1.) $\overline{ch}(t) \leq ch^{\lambda}(t)$; (2.) $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$; (3.) $s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$.

Observe that the three conditions of the definition above correspond to the requirements $(i\text{-}iii)$ we outlined at the end of the previous section. Therefore, Condition (1) is necessary for a meaningful translation of terms, while Conditions (2) and (3) are necessary

for respecting the identity of types. Using λ-channel types we can now define a mapping of ℂDuce types to ℂπ-calculus types that respects type equality.

Definition 5.2. *The interpretation function* $\{\{-\}\} : \mathscr{T}_{\mathbb{C}\text{Duce}} \to \mathscr{T}_{\mathbb{C}\pi}$ *is defined as follows*

$$\{\{\mathbf{b}\}\} = \mathbf{b} \quad \{\{\mathbf{0}\}\} = \mathbf{0} \quad \{\{\mathbf{1}\}\} = \mathbf{1} \quad \{\{\neg\tau\}\} = \neg\{\{\tau\}\}$$
$$\{\{\sigma \vee \tau\}\} = \{\{\sigma\}\} \vee \{\{\tau\}\} \quad \{\{\sigma \wedge \tau\}\} = \{\{\sigma\}\} \wedge \{\{\tau\}\}$$
$$\{\{\sigma \times \tau\}\} = \{\{\sigma\}\} \times \{\{\tau\}\} \quad \{\{\sigma \to \tau\}\} = ch\bar{}(\{\{\sigma\}\} \times ch^{\lambda}(\{\{\tau\}\})).$$

Theorem 5.3. *Let* σ *and* τ *be* ℂDuce *types. Then* $\sigma \leq \tau \iff \{\{\sigma\}\} \leq \{\{\tau\}\}$.

5.2 Incarnations of λ-Channels and Their Intuition

Possible choices for $ch^{\lambda}(t)$ are of the form $ch^{\lambda_0}(t) \wedge \varphi$ where $ch^{\lambda_0}(t) = \neg ch\bar{}(\neg t) \vee ch\bar{}(\mathbf{1})$ and φ is a constant type such that $ch\bar{}(\mathbf{0}) \leq \varphi$.

As the Condition (1) in Definition 5.1 clearly states, the λ-channel $ch^{\lambda}(t)$ essentially is $ch\bar{}(t)$ plus some extra stuff, some "garbage", that makes the other two conditions —hence type identity preservation— hold. The extra stuff that is added to $ch\bar{}(t)$ is basically given by $ch^{\lambda_0}(t)$. To understand the precise role played by this garbage, it is interesting to consider the following properties:

a. $ch^{\lambda_0}(\mathbf{0}) = \mathbf{1}$
b. $ch^{\lambda_0}(\mathbf{1}) = \neg ch\bar{}(\mathbf{0}) \vee ch\bar{}(\mathbf{1})$
c. $[\![(ch^{\lambda_0}(t) \wedge \neg ch\bar{}(t)) \wedge ch\bar{}(\mathbf{0})]\!] = \{c^s \mid t \not\leq s \;\&\; \neg t \not\leq s\} \cup \{c^{\mathbf{1}}\}$.

The first two properties say that $ch^{\lambda_0}(-)$ adds as garbage *at most* (point (a.)) everything and *at least* (point (b.)) all non-channel types plus the channel which outputs everything. In order to exactly determine which channels $ch^{\lambda_0}(t)$ adds to to $ch\bar{}(t)$ let us take out all $ch\bar{}(t)$ and consider just the channels that remained: this is exactly what $(ch^{\lambda_0}(t) \wedge \neg ch\bar{}(t)) \wedge ch\bar{}(\mathbf{0})$ does. Point (c.) states that these are all channels that can send values both inside and outside t. That is, these are all the channels for which it is not possible to predict the result of a test that checks whether the messages they transport are of type t.

This last observation is the key to understand why the complicated definition of $ch^{\lambda_0}(-)$ is necessary. We have observed that the MT translation does not work because it allows a "output-driven" overloading whereby a function can have different behaviours for different expected types of the result. The more general channel type $ch^{\lambda_0}(-)$ allows (potentially, in the types) the caller to "confuse" such output-driven functions, by sending "garbage" reply channels. Although in practice, encodings don't do that, the possibility of a output-driven function is ruled out also at the level of the types. It is like the presence of the Police in Utopia: everybody behaves well in Utopia, and the Police never works. But the presence of the Police is the visible representation of the fact the everybody behaves well.

To put it otherwise, if we take a channel that has type $ch^{\lambda}(s) \vee ch^{\lambda}(t)$, it is impossible to deduce whether it is only of type $ch^{\lambda}(s)$ or only of type $ch^{\lambda}(t)$. Even if it can transport all messages of type, say, t, it could be because the channel was in the garbage generated by $ch^{\lambda}(s)$. So λ-channels introduce some latent noise that makes it impossible to determine which output type they encode.

Although the constructor is parametric on a type φ, non-channel types play no active role in the encoding. Therefore it is reasonable (and it makes the encoding more understandable) to minimise φ (that is, $\varphi = ch^{\neg}(\mathbf{0})$) so that $[\![ch^{\lambda}(t)]\!]$ only contains channels. In particular, this choice implies that $ch^{\lambda}(\mathbf{0}) = ch^{\neg}(\mathbf{0})$ (all channels), $ch^{\lambda}(\mathbf{1}) = ch^{\neg}(\mathbf{1})$ (just the channel which outputs everything). All the development, however, is independent from this choice.

5.3 Encoding of the Terms

We describe here the mapping of $\mathbb{C}$Duce terms to $\mathbb{C}\pi$-calculus terms. What we translate are in fact typing derivations. To simplify the notation, we write e^{τ} assuming that τ is the type of e in the last step of the derivation. We use a similar convention for the immediate sub-expressions of e which are in the premises of the last applied rule. The translation is parametrised by a "continuation channel" α of type $ch^{\neg}(\{\!\{\tau\}\!\})$. For readability we decorate the channels with their types only when we restrict them and in rule (*fvar*). We also adopt the $\mathbb{C}$Duce's convention to write $x{:}\tau$ for the pattern $x \wedge \tau$. The translation also requires a straightforward translation of the patterns (it just encodes the types occurring in them) whose details are omitted.

Definition 5.4. *The translation of the expression e^{τ} on a channel α is defined by cases on the last applied typing rule:*

$$
\begin{array}{rl}
(const) & \{\!\{n^{\mathbf{b}_n}\}\!\}^{\Gamma}_{\alpha} = \overline{\alpha}(n) \\
(var) & \{\!\{x^{\tau}\}\!\}^{\Gamma,x:\tau}_{\alpha} = \overline{\alpha}(x) \\
(fvar) & \{\!\{f^{\tau}\}\!\}^{\Gamma}_{\alpha} = \overline{\alpha}(f^{\bigvee_{i\in I}(\{\!\{\sigma_i\}\!\}\times ch^{\lambda}(\{\!\{\tau_i\}\!\}))}) \qquad (\text{where } \tau = \bigwedge_{i\in I}(\sigma_i \to \tau_i)) \\
(pair) & \{\!\{(e_1^{\sigma_1}, e_2^{\sigma_2})^{\tau}\}\!\}^{\Gamma}_{\alpha} = (\nu\, a^{\{\!\{\sigma_1\}\!\}})(\nu\, b^{\{\!\{\sigma_2\}\!\}})(\{\!\{e_1^{\sigma_1}\}\!\}^{\Gamma}_{a} \parallel a(w{:}\{\!\{\sigma_1\}\!\}).(\{\!\{e_2^{\sigma_2}\}\!\}^{\Gamma}_{b} \parallel \\
 & \qquad b(h{:}\{\!\{\sigma_2\}\!\}).\overline{\alpha}(w,h))) \qquad (\text{where } \tau = \sigma_1 \times \sigma_2) \\
(appl) & \{\!\{(e_1^{\sigma\to\tau} e_2^{\sigma})^{\tau}\}\!\}^{\Gamma}_{\alpha} = (\nu\, a^{\{\!\{\sigma\to\tau\}\!\}})(\nu\, b^{\{\!\{\sigma\}\!\}})(\{\!\{e_1^{\sigma\to\tau}\}\!\}^{\Gamma}_{a} \parallel a(w{:}\{\!\{\sigma\to\tau\}\!\}).(\{\!\{e_2^{\sigma}\}\!\}^{\Gamma}_{b} \parallel \\
 & \qquad b(h{:}\{\!\{\sigma\}\!\}).\overline{w}(h,\alpha))) \\
(subs) & \{\!\{(e^{\sigma})^{\tau}\}\!\}^{\Gamma}_{\alpha} = (\nu\, a^{\{\!\{\sigma\}\!\}})(\{\!\{e^{\sigma}\}\!\}^{\Gamma}_{a} \parallel a(w{:}\{\!\{\sigma\}\!\}).\overline{\alpha}(w)) \qquad (\text{where } \sigma \le \tau) \\
(match) & \{\!\{(\mathtt{match}\ e^{\sigma}\ \mathtt{with}\ p_1 \Rightarrow e_1^{\tau_1} \mid p_2 \Rightarrow e_2^{\tau_2})^{\tau}\}\!\}^{\Gamma}_{\alpha} = \\
 & \quad (\nu\, a^{\{\!\{\sigma\}\!\}})(\nu\, b^{(\{\!\{\sigma_1\}\!\}\times ch^{\neg}(\{\!\{\tau_1\}\!\}))\vee(\{\!\{\sigma_2\}\!\}\times ch^{\neg}(\{\!\{\tau_2\}\!\}))})((P_1 + P_2) \parallel Q) \\
 & \qquad \text{where } P_1 = b(\{\!\{p_1\}\!\}, d{:}ch^{\neg}(\{\!\{\tau_1\}\!\})).\{\!\{e_1^{\tau_1}\}\!\}^{\Gamma,\sigma_1/p_1}_{d}, \\
 & \qquad\qquad P_2 = b(\{\!\{p_2 \wedge \neg\lfloor p_1 \rfloor\}\!\}, d{:}ch^{\neg}(\{\!\{\tau_2\}\!\})).\{\!\{e_2^{\tau_2}\}\!\}^{\Gamma,\sigma_2/p_2}_{d}, \\
 & \qquad\qquad Q = \{\!\{e^{\sigma}\}\!\}^{\Gamma}_{a} \parallel a(h{:}\{\!\{\sigma\}\!\}).\overline{b}(h,\alpha) \\
 & \qquad\qquad \sigma_1 = \sigma \wedge \lfloor p_1 \rfloor, \quad \sigma_2 = \sigma \wedge \neg\lfloor p_1 \rfloor, \quad \tau = \bigvee_{\{i \mid \sigma_i \not\simeq \mathbf{0}\}} \tau_i \\
(abstr) & \{\!\{(\mu f^{\bigwedge_{i\in I}(\sigma_i\to\tau_i)}(x).e)^{\tau}\}\!\}^{\Gamma}_{\alpha} = (\nu\, f^{\bigvee_{i\in I}(\{\!\{\sigma_i\}\!\}\times ch^{\lambda}(\{\!\{\tau_i\}\!\}))})(\overline{\alpha}(f) \parallel \mathsf{body}(f)) \\
 & \qquad \text{where } \mathsf{body}(f) = !(\textstyle\sum_{i\in I} f(x{:}\{\!\{\sigma_i\}\!\}, b{:}ch^{\neg}(\{\!\{\tau_i\}\!\})).\{\!\{e^{\tau_i}\}\!\}^{\Gamma,x:\sigma_i}_{b} \\
 & \qquad\qquad\qquad + f(x{:}\textstyle\bigvee_{i\in I}\{\!\{\sigma_i\}\!\}, b{:}\bigvee_{i\in I}(ch^{\lambda}(\{\!\{\tau_i\}\!\}) \wedge \neg ch^{\neg}(\{\!\{\tau_i\}\!\})).\mathsf{0}) \\
 & \qquad\qquad\qquad \tau = \textstyle\bigwedge_{i\in I}(\sigma_i \to \tau_i) \wedge \bigwedge_{j\in J} \neg(\sigma'_j \to \tau'_j).
\end{array}
$$

In rule (*fvar*), we assume that every μ-abstracted variable f has a corresponding channel constant f^t for every suitable $\mathbb{C}\pi$ type t. This allows the encoding to be parametric only in the Γ environment, and not in the Δ one.

In a match the expressions e_1 and e_2 play the role of two functions to be chosen in alternative according to the type of the argument e. Therefore we encode the match with a patterned sum of the encodings of e_1 and e_2 in parallel with the encoding of e.

The translation of a functional term is very similar to the original MT translation. To deal with overloading, the body of the function features a patterned choice. This choice includes all different behaviours that the function can produce on different inputs, and the special sub-term $f(x{:}\bigvee_{i\in I}\{\{\sigma_i\}\}, b{:}\bigvee_{i\in I}(ch^{\lambda}(\{\{\tau_i\}\}) \wedge \neg ch^{\neg}(\{\{\tau_i\}\})).\mathsf{0}$, which we call the *functional garbage*. The role of this sub-term is to obtain well-typed terms. However we will see that, within the context of translation of ℂDuce terms, the functional garbage choice is never taken. Indeed, carrying on with our analogy, this functional garbage corresponds to the prison of Utopia: it is there to capture misbehaving terms, even if we all know that there isn't any.

6 Correctness of the Encoding

We start by stating that the translation produces well-typed terms.

Theorem 6.1. *If* $\Delta;\Gamma \vdash e : \tau$*, then* $\{\{\Gamma\}\} \vdash \{\{e^{\tau}\}\}^{\Gamma}_{c:\{\{\tau\}\}}$ *and* $\{\{\Gamma\}\}, x : ch^{\neg}(\{\{\tau\}\}) \vdash \{\{e^{\tau}\}\}^{\Gamma}_{x}$*, where* $\{\{\Gamma\}\} = \{y : \{\{\sigma\}\} \mid y : \sigma \in \Gamma\}$.

In the following we convene that when we write $\{\{e\}\}^{\Gamma}_{c}$, then there are τ and Δ such that $\Delta;\Gamma \vdash e : \tau$ and $ch^{\neg}(\{\{\tau\}\})$ is the type of c.

A first observation is that all reductions out of the encoding of a ℂDuce expression are deterministic (since patterns in sums are mutually exclusive) and never use the functional garbage in the body of functions. A *functional redex* is a redex of the shape $\mathsf{body}(f) \parallel \overline{f}(v,c)$. A reduction is *safe* if it is deterministic and each functional redex is reduced by choosing an alternative in $\mathsf{body}(f)$ different from the functional garbage. We denote safe reductions by $\longrightarrow_s$: as usual $\longrightarrow_s^*$ is the reflexive and transitive closure of $\longrightarrow_s$.

Lemma 6.2. *All reductions starting from* $\{\{e\}\}^{\varnothing}_{c}$ *where e is an arbitrary* ℂDuce *expression are safe.*

In order to state the correctness of the encoding, it is crucial to understand how ℂDuce values are mapped to ℂπ processes. As it is clear from the encoding, a functional value is mapped into the output of a private channel name in parallel with the encoding of the function body. We can then say that the ℂπ value corresponding to a functional value is a channel name. The encoding of a pair of ℂDuce values reduces to a process which outputs the pair of the corresponding ℂπ values in parallel with the function bodies of all functions which occur in the two values.

To formalise the above we will assume that *all function names* in the current value are *distinct* and *fixed*, so that we cannot rename them. We define two mappings, one from ℂDuce values to ℂπ values and one from ℂDuce values to sets of channel names.

Definition 6.3

1. *The mapping* $\mathsf{cpv}(-)$ *is defined by induction on* $\mathbb{C}$Duce *values as follows:*

- $\mathsf{cpv}(n) = n$;
- $\mathsf{cpv}(\mu f \bigwedge_{i\in I}(\sigma_i \to \tau_i)(x).e) = f^{\bigvee_{i\in I}(\{\{\sigma_i\}\} \times ch^{\lambda}(\{\{\tau_i\}\}))}$;
- $\mathsf{cpv}((v_1, v_2)) = (\mathsf{cpv}(v_1), \mathsf{cpv}(v_2))$.

2. *The mapping* $\mathsf{func}(-)$ *is defined by induction on* $\mathbb{C}$Duce *values as follows:*

- $\mathsf{func}(n) = \varnothing$;
- $\mathsf{func}(\mu f \bigwedge_{i\in I}(\sigma_i \to \tau_i)(x).e) = \{f^{\bigvee_{i\in I}(\{\{\sigma_i\}\} \times ch^{\lambda}(\{\{\tau_i\}\}))}\}$;
- $\mathsf{func}((v_1, v_2)) = \mathsf{func}(v_1) \cup \mathsf{func}(v_2)$.

Let $\mathsf{body}(f)$ be defined as in the last clause of Definition 5.4, then the above mappings can express the normal forms of processes encoding values:

Lemma 6.4. $\{\{v\}\}_c^{\varnothing} \longrightarrow_s^* (\nu\, \mathsf{func}(v))(\overline{c}(\mathsf{cpv}(v)) \| _{f\in \mathsf{func}(v)} \mathsf{body}(f))$.

More generally, one would like to have that if e is a well-typed $\mathbb{C}$Duce expression and $e \longrightarrow^* v$, then $\{\{e\}\}_c^{\varnothing} \longrightarrow_s^* (\nu\, \mathsf{func}(v))(\overline{c}(\mathsf{cpv}(v)) \| _{f\in \mathsf{func}(v)} \mathsf{body}(f))$. Unfortunately, the corresponding result does not even hold for the MT encoding of λ-calculus into π-calculus [15], *a fortiori* nor does for our encoding.

Our encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$ being essentially an extension of the MT encoding has luckily no more problems than the original one, so we can show similar soundness results. To formulate these results we need to define for $\mathbb{C}\pi$ processes a standard notion of typed barbed congruence with respect to an environment Γ ($\Gamma \rhd P \cong Q$), see [17]. The main theorem of this section states that if a $\mathbb{C}$Duce expression reduces to a value, then its encoding reduces to a process which is barbed congruent to the normal form of the encoding of that value, and vice versa if the evaluation of a $\mathbb{C}$Duce expression does not terminate, then the evaluation of its encoding does not terminate either.

Theorem 6.5 (Correctness). *If* $e \longrightarrow^* v$, *then* $\{\{e\}\}_c^{\varnothing} \longrightarrow_s^* P$ *for some* P *such that* $\varnothing \rhd P \cong (\nu\, \mathsf{func}(v))(\overline{c}(\mathsf{cpv}(v)) \| _{f\in \mathsf{func}(v)} \mathsf{body}(f))$. *If* e *diverges, then so does* $\{\{e\}\}_c^{\varnothing}$.

From this, and from compositionality, it is easy to obtain soundness. Given two $\mathbb{C}$Duce terms $\Delta;\Gamma \vdash e : \tau$ and $\Delta;\Gamma \vdash e' : \tau$ we denote by $\Delta;\Gamma \rhd e \approx e'$ the standard Morris-style observational congruence (as defined, for instance, in [17] pag. 478).

Corollary 6.6 (Soundness). *If* $\Delta;\Gamma \vdash e : \tau$ *and* $\Delta;\Gamma \vdash e' : \tau$ *and* $\{\{\Gamma\}\} \rhd \{\{e\}\}_c^{\Gamma} \cong \{\{e'\}\}_c^{\Gamma}$, *then* $\Delta;\Gamma \rhd e \approx e'$.

Notice that completeness fails for our encoding, for the same reason as it fails for the original MT encoding.

7 Conclusion

In this paper we presented a localised version of the $\mathbb{C}\pi$-calculus which allows for fully recursive types, on top of the already rich type structure of $\mathbb{C}\pi$. We then showed how this can be used to type-faithfully encode $\mathbb{C}$Duce.

If we merely stop at the technical result, then the interest of this work is quite limited: sure, it shows the correspondence between overloading and guarded choices; sure, this can be seen as the work that paves the way toward a concrete implementation of a concurrent programming language based on ℂDuce, similarly to the way the JoCaml language was derived from OCaml and Join. But again this would look as some solid, technically impeccable, and extremely boring achievement.

However, we think that the added value of this work lies more in the lessons we learnt and the techniques we developed, than directly in its result.

Foremost, we learnt that the process that encodes a function has much more power than the function it encodes. This is because it has more elements to work on, both the argument and the return channel, and it is thus characterised by a wider spectrum of possible choices. This looks bluntly obvious, worthy of Monsieur De La Palice's troops, but note that this aspect was totally hidden in all previous encodings. Indeed this is emphasised only by the presence of linguistic branching constructs for which the type system must cover all alternatives. This is the case of pattern matching, where the pattern exhaustiveness requirement forces types to take into account all possible combinations.

This situation requires the introduction of some noise at the level of the types in order to compensate for the asymmetry between the caller of the function (the service client) and the executor of the function (the service server). This technique could be seen as a security policy that the client implements at type level to defend itself from possible misbehaviour of the server. The client performs a type obfuscation: in this way it reserves for itself the possibility to send rogue arguments and so it threatens the server against misbehaviour. We hope that these techniques of type obfuscation could be generalised to various security scenarios and we aim to explore them in the future.

As noted, the Milner-Turner encoding bears strong resemblance with the continuation passing style (CPS) techniques used in functional programming. All the above observations can be indeed carried over to such framework. Using these intuitions, we plan to study CPS transforms for ℂDuce. This should have a very important practical impact: ℂDuce (we mean, the implemented language) was recently extended to deal with Web-services and active Web pages, and we consider CPS as the key technique to implement stateless Web sessions on the top of them.

The other important aspect of this work is that it constitutes an independent, though indirect, confirmation that ℂπ yields the right equational theory of union and intersection types for the π-calculus. Pierce and Sangiorgi's subtyping for the π-calculus, though very elegant, is structurally very poor: it essentially amounts to compare the levels of nesting of channel constructors with the same polarity. In order to obtain a much richer and expressive subtyping relation, one can resort to union and intersection types. However, the problem arises on which equational theory to use for these types. ℂπ gives a precise and semantically grounded answer for it (and for negation types): its semantic justification for the equational theory, and its correspondence with set-theory constitute a first strong justification for it.

The equational theory of ℂπ is partially justified in practice, since works such as the PiDuce project carried out at the University of Bologna [4] and the language XPi developed at the University of Marseille [1], feature restrictions of the ℂπ type system

that fit XML data manipulation. The present work is another, more theoretical, confirmation of the validity of the $\mathbb{C}\pi$ theory. If we admit that the Milner-Turner encoding is very natural, then we see how perfectly the laws of $\mathbb{C}\pi$ fit the MT encoding, stressing the asymmetry of the roles of client and server, and pushing the emergence of the type obfuscation technique. This is what we consider the most important achievement of this work.

References

1. L. Acciai and M. Boreale. XPi: A typed process calculus for XML messaging. In *FMOODS*, volume 3535 of *LNCS*, pages 47–66. Springer-Verlag, 2005.
2. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03*, pages 51–63. ACM Press, 2003.
3. G. Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, `http://www.inria.fr/rrrt/rr-1702.html`, 1992.
4. A. L. Brown, C. Laneve, and L. G. Meredith. PiDuce: A process calculus with native XML datatypes. In *EPEW/WS-FM*, volume 3670 of *LNCS*, pages 18–34. Springer-Verlag, 2005.
5. G. Castagna. Semantic subtyping: challenges, perspectives, and open problems. In *ICTCS 2005*, volume 3701 of *LNCS*, pages 1–20. Springer-Verlag, 2005.
6. G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *LICS '05*, pages 92–101. IEEE Computer Society Press, 2005.
7. G. Castagna, M. Dezani-Ciancaglini, and D. Varacca. Encoding $\mathbb{C}$Duce in the $\mathbb{C}\pi$-calculus. Extended version, `http://www.di.unito.it/~dezani/papers/cdv.pdf`, 2006.
8. G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In *PPDP '05, ACM Press* (full version) and *ICALP '05, LNCS* volume 3580, Springer-Verlag (summary), 2005. Joint ICALP-PPDP keynote talk.
9. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.
10. A. Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, Université Paris 7, 2004.
11. A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *LICS '02*, pages 137–146. IEEE Computer Society Press, 2002.
12. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
13. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP 91*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.
14. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *ICALP'98*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, 1998.
15. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
16. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
17. D. Sangiorgi and D. Walker. *The π-calculus*. Cambridge University Press, 2002.
18. N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the π-Calculus. *Information and Computation*, 191(2):145–202, 2004.