Realizability interpretation and normalization of typed call-by-need λ-calculus with control

Étienne Miquey[1,2] and Hugo Herbelin[2]

[1] Équipe Gallinette
Inria, LS2N (CNRS)
Université de Nantes
[2] Équipe πr^2
Inria, IRIF (CNRS)
Université Paris-Diderot
etienne.miquey,herbelin@inria.fr

Abstract. We define a variant of Krivine realizability where realizers are pairs of a term and a substitution. This variant allows us to prove the normalization of a simply-typed call-by-need λ-calculus with control due to Ariola *et al.* Indeed, in such call-by-need calculus, substitutions have to be delayed until knowing if an argument is really needed. We then extend the proof to a call-by-need λ-calculus equipped with a type system equivalent to classical second-order predicate logic, representing one step towards proving the normalization of the call-by-need classical second-order arithmetic introduced by the second author to provide a proof-as-program interpretation of the axiom of dependent choice.

Introduction

Realizability-based normalization

Normalization by realizability is a standard technique to prove the normalization of typed λ-calculi. Originally introduced by Tait [36] to prove the normalization of System T, it was extended by Girard to prove the normalization of System F [11]. This kind of techniques, also called normalization by reducibility or normalization by logical relations, works by interpreting each type by a set of typed or untyped terms seen as realizers of the type, then showing that the way these sets of realizers are built preserve properties such as normalization. Over the years, multiple uses and generalization of this method have been done, for a more detailed account of which we refer the reader to the work of Gallier [9].

Realizability techniques were adapted to the normalization of various calculi for classical logic (see e.g. [3, 32]). A specific framework tailored to the study of realizability for classical logic has been designed by Krivine [22] on top of a λ-calculus with control whose reduction is defined in terms of an abstract machine. In such a machinery, terms are evaluated in front of stacks; and control (thus classical logic) is made available through the possibility of saving and restoring stacks. During the last twenty years, Krivine's classical realizability turned out

to be fruitful both from the point of view of logic, leading to the construction of new models of set theory, and generalizing in particular the technique of Cohen's forcing [18–20]; and on its computational facet, providing alternative tools to the analysis of the computational content of classical programs[3].

Noteworthily, Krivine realizability is one of the approaches contributing to advocating the motto that through the Curry-Howard correspondence, with new programming instructions come new reasoning principles[4]. Our original motivation for the present work is actually in line with this idea, in the sense that our long-term purpose is to give a realizability interpretation to dPA^ω, a call-by-need calculus defined by the second author [15]. In this calculus, the lazy evaluation is indeed a fundamental ingredient in order to obtain an executable proof term for the axiom of dependent choice.

Contributions of the paper

In order to address the normalization of typed call-by-need λ-calculus, we design a variant of Krivine's classical realizability, where the realizers are closures (a term with a substitution for its free variables). The call-by-need λ-calculus with control that we consider is the $\overline{\lambda}_{[lv\tau\star]}$-calculus. This calculus, that was defined by Ariola *et al.* [2], is syntactically described in an extension with explicit substitutions of the $\lambda\mu\tilde{\mu}$-calculus [6, 14, 29]. The syntax of the $\lambda\mu\tilde{\mu}$-calculus itself refines the syntax of the λ-calculus by syntactically distinguishing between *terms* and *evaluation contexts*. It also contains *commands* which combine terms and evaluation contexts so that they can interact together. Thinking of evaluation contexts as stacks and commands as states, the $\lambda\mu\tilde{\mu}$-calculus can also be seen as a syntax for abstract machines. As for a proof-as-program point of view, the $\lambda\mu\tilde{\mu}$-calculus and its variants can be seen as a term syntax for proofs of Gentzen's sequent calculus. In particular, the $\lambda\mu\tilde{\mu}$-calculus contains control operators which give a computational interpretation to classical logic.

We give a proof of normalization first for the simply-typed $\overline{\lambda}_{[lv\tau\star]}$-calculus[5], then for a type system with first-order and second-order quantification. While we only apply our technique to the normalization of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, our interpretation incidentally suggests a way to adapt Krivine realizability to other call-by-need settings. This paves the way to the computational interpretation of classical proofs using lazy evaluation or shared memory cells, including the case of the call-by-need second order arithmetic dPA^ω [15].

[3] See for instance [27] about witness extraction or [12, 13] about specification problems.

[4] For instance, one way to realize the axiom of dependent choice in classical realizability is by means of an extra instruction `quote` [17].

[5] Even though it has not been done formally, the normalization of the $\overline{\lambda}_{lv}$-calculus presented in [2] should also be derivable from Polonowski's proof of strong normalization of the non-deterministic $\lambda\mu\tilde{\mu}$-calculus [35]. The $\overline{\lambda}_{lv}$-calculus (a big-step variant of the $\overline{\lambda}_{[lv\tau\star]}$-calculus introduced in Ariola *et al.*) is indeed a particular evaluation strategy for the $\lambda\mu\tilde{\mu}$-calculus, so that the strong normalization of the non-deterministic variant of the latter should imply the normalization of the former as a particular case.

1 The $\overline{\lambda}_{[lv\tau\star]}$-calculus

1.1 The call-by-need evaluation strategy

The call-by-need evaluation strategy of the λ-calculus evaluates arguments of functions only when needed, and, when needed, shares their evaluations across all places where the argument is required. The call-by-need evaluation is at the heart of a functional programming language such as Haskell. It has in common with the call-by-value evaluation strategy that all places where a same argument is used share the same value. Nevertheless, it observationally behaves like the call-by-name evaluation strategy (for the pure λ-calculus), in the sense that a given computation eventually evaluates to a value if and only if it evaluates to the same value (up to inner reduction) along the call-by-name evaluation. In particular, in a setting with non-terminating computations, it is not observationally equivalent to the call-by-value evaluation. Indeed, if the evaluation of a useless argument loops in the call-by-value evaluation, the whole computation loops, which is not the case of call-by-name and call-by-need evaluations.

These three evaluation strategies can be turned into equational theories. For call-by-name and call-by-value, this was done by Plotkin through continuation-passing-style (CPS) semantics characterizing these theories [34]. For the call-by-need evaluation strategy, a specific equational theory reflecting the intensional behavior of the strategy into a semantics was proposed independently by Ariola and Felleisen [1], and by Maraist, Odersky and Wadler [26]. A continuation-passing-style semantics was proposed in the 90s by Okasaki, Lee and Tarditi [30]. However, this semantics does not ensure normalization of simply-typed call-by-need evaluation, as shown in [2], thus failing to ensure a property which holds in the simply-typed call-by-name and call-by-value cases.

Continuation-passing-style semantics *de facto* gives a semantics to the extension of λ-calculus with control operators[6]. In particular, even though call-by-name and call-by-need are observationally equivalent on pure λ-calculus, their different intentional behaviors induce different CPS semantics, leading to different observational behaviors when control operators are considered. On the other hand, the semantics of calculi with control can also be reconstructed from an analysis of the duality between programs and their evaluation contexts, and the duality between the `let` construct (which binds programs) and a control operator such as Parigot's μ (which binds evaluation contexts). Such an analysis can be done in the context of the $\lambda\mu\tilde{\mu}$-calculus [6, 14].

In the call-by-name and call-by-value cases, the approach based on $\lambda\mu\tilde{\mu}$-calculus leads to continuation-passing style semantics similar to the ones given by Plotkin or, in the call-by-name case, also to the one by Lafont, Reus and Streicher [23]. As for call-by-need, in [2] is defined the $\overline{\lambda}_{lv}$-calculus, a call-by-need version of the $\lambda\mu\tilde{\mu}$-calculus. A continuation-passing style semantics is then defined via a calculus called $\overline{\lambda}_{[lv\tau\star]}$ [2]. This semantics, which is different from Okasaki, Lee and Tarditi's one [30], is the object of study in this paper.

[6] That is to say with operators such as Scheme's `callcc`, Felleisen's $\mathcal{C}$, $\mathcal{K}$, or $\mathcal{A}$ operators [8], Parigot's μ and [ ] operators [31], Crolard's `catch` and `throw` operators [5].

1.2 Explicit environments

While the results presented in this paper could be directly expressed using the $\overline{\lambda}_{lv}$-calculus, the realizability interpretation naturally arises from the decomposition of this calculus into a different calculus with an explicit *environment*, the $\overline{\lambda}_{[lv\tau\star]}$-calculus [2]. Indeed, as we shall see in the sequel, the decomposition highlights different syntactic categories that are deeply involved in the type system and in the definition of the realizability interpretation.

The $\overline{\lambda}_{[lv\tau\star]}$-calculus is a reformulation of the $\overline{\lambda}_{lv}$-calculus with explicit environments, called *stores* and denoted by τ. Stores consists of a list of bindings of the form $[x := t]$, where x is a term variable and t a term, and of bindings of the form $[\alpha := e]$ where α is a context variable and e a context. For instance, in the closure $c\tau[x := t]\tau'$, the variable x is bound to t in c and τ'. Besides, the term t might be an unevaluated term (*i.e.* lazily stored), so that if x is eagerly demanded at some point during the execution of this closure, t will be reduced in order to obtain a value. In the case where t indeed produces a value V, the store will be updated with the binding $[x := V]$. However, a binding of this form (with a value) is fixed for the rest of the execution. As such, our so-called stores somewhat behave like lazy explicit substitutions or mutable environments.

To draw the comparison between our structures and the usual notions of stores and environments, two things should be observed. First, the usual notion of store refers to a structure of list that is fully mutable, in the sense that the cells can be updated at any time and thus values might be replaced. Second, the usual notion of environment designates a structure in which variables are bounded to closures made of a term and an environment. In particular, terms and environments are duplicated, *i.e.* sharing is not allowed. Such a structure resemble to a tree whose nodes are decorated by terms, as opposed to a machinery allowing sharing (like ours) whose underlying structure is broadly a directed acyclic graph. See for instance [24] for a Krivine abstract machine with sharing.

1.3 Syntax & reduction rules

The lazy evaluation of terms allows for the following reduction rule: us to reduce a command $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle$ to the command c' together with the binding $[x := \mu\alpha.c]$.

$$\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle \to c'[x := \mu\alpha.c]$$

In this case, the term $\mu\alpha.c$ is left unevaluated ("frozen") in the store, until possibly reaching a command in which the variable x is needed. When evaluation reaches a command of the form $\langle x \| F \rangle \tau[x := \mu\alpha.c]\tau'$, the binding is opened and the term is evaluated in front of the context $\tilde{\mu}[x].\langle x \| F \rangle \tau'$:

$$\langle x \| F \rangle \tau[x := \mu\alpha.c]\tau' \to \langle \mu\alpha.c \| \tilde{\mu}[x].\langle x \| F \rangle \tau' \rangle \tau$$

The reader can think of the previous rule as the "defrosting" operation of the frozen term $\mu\alpha.c$: this term is evaluated in the prefix of the store τ which predates it, in front of the context $\tilde{\mu}[x].\langle x \| F \rangle \tau'$ where the $\tilde{\mu}[x]$ binder is waiting for a value.

Strong values	v	$::= \lambda x.t \mid \mathbf{k}$
Weak values	V	$::= v \mid x$
Terms	t, u	$::= V \mid \mu\alpha.c$
Forcing contexts	F	$::= t \cdot E \mid \boldsymbol{\kappa}$
Catchable contexts	E	$::= F \mid \alpha \mid \tilde{\mu}[x].\langle x \| F\rangle\tau$
Evaluation contexts	e	$::= E \mid \tilde{\mu}x.c$
Stores	τ	$::= \varepsilon \mid \tau[x := t] \mid \tau[\alpha := E]$
Commands	c	$::= \langle t \| e\rangle$
Closures	l	$::= c\tau$

(Beta)	$\langle \lambda x.t \| u \cdot E\rangle\tau$	$\to$	$\langle u \| \tilde{\mu}x.\langle t \| E\rangle\rangle\tau$
(Let)	$\langle t \| \tilde{\mu}x.c\rangle\tau$	$\to$	$c\tau[x := t]$
(Catch)	$\langle \mu\alpha.c \| E\rangle\tau$	$\to$	$c\tau[\alpha := E]$
(Lookup$_\alpha$)	$\langle V \| \alpha\rangle\tau[\alpha := E]\tau'$	$\to$	$\langle V \| E\rangle\tau[\alpha := E]\tau'$
(Lookup$_x$)	$\langle x \| F\rangle\tau[x := t]\tau'$	$\to$	$\langle t \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau$
(Restore)	$\langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau$	$\to$	$\langle V \| F\rangle\tau[x := V]\tau'$

Fig. 1. Syntax and reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

This context keeps trace of the part of the store τ' that was originally located after the binding $[x := ...]$. This way, if a value V is indeed furnished for the binder $\tilde{\mu}[x]$, the original command $\langle x \| F\rangle$ is evaluated in the updated full store:

$$\langle V \| \tilde{\mu}[x].\langle x \| F\rangle\tau'\rangle\tau \to \langle V \| F\rangle\tau[x := V]\tau'$$

The brackets in $\tilde{\mu}[x].c$ are used to express the fact that the variable x is forced at top-level (unlike contexts of the shape $\tilde{\mu}x.C[\langle x \| F\rangle]$ in the $\overline{\lambda}_{lv}$-calculus). The reduction system resembles the one of an abstract machine. Especially, it allows us to keep the standard redex at the top of a command and avoids searching through the meta-context for work to be done.

Note that our approach slightly differ from [2] since we split values into two categories: strong values (v) and weak values (V). The strong values correspond to values strictly speaking. The weak values include the variables which force the evaluation of terms to which they refer into shared strong value. Their evaluation may require capturing a continuation. The syntax of the language, which includes constants $\mathbf{k}$ and co-constants $\boldsymbol{\kappa}$, is given in Figure 1. As for the reduction $\to$, we define it as the compatible reflexive transitive closure of the rules given in Figure 1.

The different syntactic categories can be understood as the different levels of alternation in a context-free abstract machine (see [2]): the priority is first given to contexts at level e (lazy storage of terms), then to terms at level t (evaluation of $\mu\alpha$ into values), then back to contexts at level E and so on until level v. These different categories are directly reflected in the definition of the abstract machine defined in [2], and will thus be involved in the definition of our realizability interpretation. We chose to highlight this by distinguishing different types of sequents already in the typing rules that we shall now present.

$$\frac{(\mathbf{k} : X) \in \mathcal{S}}{\Gamma \vdash_v \mathbf{k} : X}\,(\mathbf{k}) \qquad \frac{\Gamma, x : A \vdash_t t : B}{\Gamma \vdash_v \lambda x.t : A \to B}\,(\to_r) \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash_V x : A}\,(x) \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}\,(\uparrow^V)$$

$$\frac{(\boldsymbol{\kappa} : A) \in \mathcal{S}}{\Gamma \vdash_F \boldsymbol{\kappa} : A^\perp}\,(\boldsymbol{\kappa}) \qquad \frac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_E E : B^\perp}{\Gamma \vdash_F t \cdot E : (A \to B)^\perp}\,(\to_l) \qquad \frac{(\alpha : A) \in \Gamma}{\Gamma \vdash_E \alpha : A^\perp}\,(\alpha)$$

$$\frac{\Gamma \vdash_F F : A^\perp}{\Gamma \vdash_E F : A^\perp}\,(\uparrow^E) \qquad \frac{\Gamma \vdash_V V : A}{\Gamma \vdash_t V : A}\,(\uparrow^t) \qquad \frac{\Gamma, \alpha : A^\perp \vdash_c c}{\Gamma \vdash_t \mu\alpha.c : A}\,(\mu) \qquad \frac{\Gamma \vdash_E E : A^\perp}{\Gamma \vdash_e E : A^\perp}\,(\uparrow^e)$$

$$\frac{\Gamma, x : A \vdash_c c}{\Gamma \vdash_e \tilde{\mu}x.c : A^\perp}\,(\tilde{\mu}) \qquad \frac{\Gamma, x : A, \Gamma' \vdash_F F : A^\perp \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_E \tilde{\mu}[x].\langle x \| F \rangle \tau : A^\perp}\,(\tilde{\mu}^{[]})$$

$$\frac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_e e : A^\perp}{\Gamma \vdash_c \langle t \| e \rangle}\,(c) \qquad \frac{\Gamma, \Gamma' \vdash_c c \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}\,(l) \qquad \frac{}{\Gamma \vdash_\tau \varepsilon : \varepsilon}\,(\varepsilon)$$

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \quad \Gamma, \Gamma' \vdash_t t : A}{\Gamma \vdash_\tau \tau[x := t] : \Gamma', x : A}\,(\tau_t) \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma' \quad \Gamma, \Gamma' \vdash_E E : A^\perp}{\Gamma \vdash_\tau \tau[\alpha := E] : \Gamma', \alpha : A^\perp}\,(\tau_E)$$

Fig. 2. Typing rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

1.4 A type system for the $\overline{\lambda}_{[lv\tau\star]}$-calculus.

We have nine kinds of (one-sided) sequents, one for typing each of the nine syntactic categories. We write them with an annotation on the $\vdash$ sign, using one of the letters v, V, t, F, E, e, l, c, τ. Sequents typing values and terms are asserting a type, with the type written on the right; sequents typing contexts are expecting a type A with the type written $A^\perp$; sequents typing commands and closures are black boxes neither asserting nor expecting a type; sequents typing substitutions are instantiating a typing context. In other words, we have the following nine kinds of sequents:

$$\begin{array}{lll} \Gamma \vdash_l l & \Gamma \vdash_t t : A & \Gamma \vdash_e e : A^\perp \\ \Gamma \vdash_c c & \Gamma \vdash_V V : A & \Gamma \vdash_E E : A^\perp \\ \Gamma \vdash_\tau \tau : \Gamma' & \Gamma \vdash_v v : A & \Gamma \vdash_F F : A^\perp \end{array}$$

where types and typing contexts are defined by:

$$A, B ::= X \mid A \to B \qquad\qquad \Gamma ::= \varepsilon \mid \Gamma, x : A \mid \Gamma, \alpha : A^\perp$$

The typing rules are given on Figure 2 where we assume that a variable x (resp. co-variable α) only occurs once in a context Γ (we implicitly assume the possibility of renaming variables by α-conversion). We also adopt the convention that constants $\mathbf{k}$ and co-constants $\boldsymbol{\kappa}$ come with a signature $\mathcal{S}$ which assigns them a type. This type system enjoys the property of subject reduction.

Theorem 1 (Subject reduction). *If $\Gamma \vdash_l c\tau$ and $c\tau \to c'\tau'$ then $\Gamma \vdash_l c'\tau'$.*

Proof. By induction on typing derivations. □

2 Normalization of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

2.1 Normalization by realizability

The proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus that we present in this section is inspired from techniques of Krivine's classical realizability [22], whose notations we borrow. Actually, it is also very close to a proof by reducibility[7]. In a nutshell, to each type A is associated a set $|A|_t$ of terms whose execution is guided by the structure of A. These terms are the ones usually called *realizers* in Krivine's classical realizability. Their definition is in fact indirect, and is done by orthogonality to a set of "correct" computations, called a *pole*. The choice of this set is central when studying models induced by classical realizability for second-order-logic, but in the present case we only pay attention to the particular pole of terminating computations. This is where lies one of the difference with usual proofs by reducibility, where everything is done with respect to SN, while our definition are parametric in the pole (which is chosen to be SN in the end). The adequacy lemma, which is the central piece, consists in proving that typed terms belong to the corresponding sets of realizers, and are thus normalizing.

More in details, our proof can be sketched as follows. First, we generalize the usual notion of closed term to the notion of closed *term-in-store*. Intuitively, this is due to the fact that we are no longer interested in closed terms and substitutions to close opened terms, but rather in terms that are closed when considered in the current store. This is based on the simple observation that a store is nothing more than a shared substitution whose content might evolve along the execution. Second, we define the notion of *pole* $\perp\!\!\!\perp$, which are sets of closures closed by anti-evaluation and store extension. In particular, the set of normalizing closures is a valid pole. This allows to relate terms and contexts thanks to a notion of orthogonality with respect to the pole. We then define for each formula A and typing level o (of e, t, E, V, F, v) a set $|A|_o$ (resp. $\|A\|_o$) of terms (resp. contexts) in the corresponding syntactic category. These sets correspond to reducibility candidates, or to what is usually called truth values and falsity values in Krivine realizability. Finally, the core of the proof consists in the adequacy lemma, which shows that any closed term of type A at level o is in the corresponding set $|A|_o$. This guarantees that any typed closure is in any pole, and in particular in the pole of normalizing closures. Technically, the proof of adequacy evaluates in each case a state of an abstract machine (in our case a closure), so that the proof also proceeds by evaluation. A more detailed explanation of this observation as well as a more introductory presentation of normalization proofs by classical realizability are given in an article by Dagand and Scherer [7].

2.2 Realizability interpretation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

We begin by defining some key notions for stores that we shall need further in the proof.

[7] See for instance the proof of normalization for system D presented in [21, 3.2].

Definition 2 (Closed store). *We extend the notion of free variable to stores:*

$$\begin{array}{rcl} FV(\varepsilon) & \triangleq & \emptyset \\ FV(\tau[x := t]) & \triangleq & FV(\tau) \cup \{y \in FV(t) : y \notin \mathtt{dom}(\tau)\} \\ FV(\tau[\alpha := E]) & \triangleq & FV(\tau) \cup \{\beta \in FV(E) : \beta \notin \mathtt{dom}(\tau)\} \end{array}$$

so that we can define a closed store *to be a store τ such that $FV(\tau) = \emptyset$.*

Definition 3 (Compatible stores). *We say that two stores τ and τ' are* independent *and write $\tau \# \tau'$ when $\mathtt{dom}(\tau) \cap \mathtt{dom}(\tau') = \emptyset$. We say that they are* compatible *and write $\tau \diamond \tau'$ whenever for all variables x (resp. co-variables α) present in both stores: $x \in \mathtt{dom}(\tau) \cap \mathtt{dom}(\tau')$; the corresponding terms (resp. contexts) in τ and τ' coincide. Finally, we say that τ' is an* extension *of τ and write $\tau \lhd \tau'$ whenever $\mathtt{dom}(\tau) \subseteq \mathtt{dom}(\tau')$ and $\tau \diamond \tau'$.*

We denote by $\overline{\tau\tau'}$ the compatible union $\mathtt{join}(\tau, \tau')$ of closed stores τ and τ', defined by:

$$\begin{array}{rcll} \mathtt{join}(\tau_0[x := t]\tau_1, \tau_0'[x := t]\tau_1') & \triangleq & \tau_0\tau_0'[x := t]\mathtt{join}(\tau_1, \tau_1') & (\text{if } \tau_0 \# \tau_0') \\ \mathtt{join}(\tau, \tau') & \triangleq & \tau\tau' & (\text{if } \tau \# \tau') \\ \mathtt{join}(\varepsilon, \tau) & \triangleq & \tau & \\ \mathtt{join}(\tau, \varepsilon) & \triangleq & \tau & \end{array}$$

The following lemma (which follows easily from the previous definition) states the main property we will use about union of compatible stores.

Lemma 4. *If τ and τ' are two compatible stores, then $\tau \lhd \overline{\tau\tau'}$ and $\tau' \lhd \overline{\tau\tau'}$. Besides, if τ is of the form $\tau_0[x := t]\tau_1$, then $\overline{\tau\tau'}$ is of the form $\tau_2[x := t]\tau_3$ with $\tau_0 \lhd \tau_2$ and $\tau_1 \lhd \tau_3$.*

Proof. This follows easily from the previous definition. □

As we explained in the introduction of this section, we will not consider closed terms in the usual sense. Indeed, while it is frequent in the proofs of normalization (*e.g.* by realizability or reducibility) of a calculus to consider only closed terms and to perform substitutions to maintain the closure of terms, this only makes sense if it corresponds to the computational behavior of the calculus. For instance, to prove the normalization of $\lambda x.t$ in typed call-by-name $\lambda\mu\tilde{\mu}$-calculus, one would consider a substitution ρ that is suitable for with respect to the typing context Γ, then a context $u \cdot e$ of type $A \to B$, and evaluates :

$$\langle \lambda x.t_\rho \| u \cdot e \rangle \quad \to \quad \langle t_\rho[u/x] \| e \rangle$$

Then we would observe that $t_\rho[u/x] = t_{\rho[x:=u]}$ and deduce that $\rho[x := u]$ is suitable for $\Gamma, x : A$, which would allow us to conclude by induction.

However, in the $\overline{\lambda}_{[lv\tau\star]}$-calculus we do not perform global substitution when reducing a command, but rather add a new binding $[x := u]$ in the store:

$$\langle \lambda x.t \| u \cdot E \rangle \tau \quad \to \quad \langle t \| E \rangle \tau[x := u]$$

Therefore, the natural notion of closed term invokes the closure under a store, which might evolve during the rest of the execution (this is to contrast with a substitution).

Definition 5 (Term-in-store). *We call* closed term-in-store *(resp.* closed context-in-store*,* closed closures*) the combination of a term t (resp. context e, command c) with a closed store τ such that $FV(t) \subseteq \mathtt{dom}(\tau)$. We use the notation $(t|\tau)$ (resp. $(e|\tau), (c|\tau)$) to denote such a pair.*

We should note that in particular, if t is a closed term, then $(t|\tau)$ is a term-in-store for any closed store τ. The notion of closed term-in-store is thus a generalization of the notion of closed terms, and we will (ab)use of this terminology in the sequel. We denote the sets of closed closures by $\mathcal{C}_0$, and will identify $(c|\tau)$ and the closure $c\tau$ when c is closed in τ. Observe that if $c\tau$ is a closure in $\mathcal{C}_0$ and τ' is a store extending τ, then $c\tau'$ is also in $\mathcal{C}_0$. We are now equipped to define the notion of pole, and verify that the set of normalizing closures is indeed a valid pole.

Definition 6 (Pole). *A subset $\perp\!\!\!\perp \subseteq \mathcal{C}_0$ is said to be* saturated *or* closed by anti-reduction *whenever for all $(c|\tau), (c'|\tau') \in \mathcal{C}_0$, if $c'\tau' \in \perp\!\!\!\perp$ and $c\tau \to c'\tau'$ then $c\tau \in \perp\!\!\!\perp$. It is said to be* closed by store extension *if whenever $c\tau \in \perp\!\!\!\perp$, for any store τ' extending τ: $\tau \lhd \tau'$, $c\tau' \in \perp\!\!\!\perp$. A* pole *is defined as any subset of $\mathcal{C}_0$ that is closed by anti-reduction and store extension.*

The following proposition is the one supporting the claim that our realizability proof is almost a reducibility proof whose definitions have been generalized with respect to a pole instead of the fixed set SN.

Proposition 7. *The set $\perp\!\!\!\perp_{\Downarrow} = \{c\tau \in \mathcal{C}_0 : \ c\tau$ normalizes $\}$ is a pole.*

Proof. As we only considered closures in $\mathcal{C}_0$, both conditions (closure by anti-reduction and store extension) are clearly satisfied:

- if $c\tau \to c'\tau'$ and $c'\tau'$ normalizes, then $c\tau$ normalizes too;
- if c is closed in τ and $c\tau$ normalizes, if $\tau \lhd \tau'$ then $c\tau'$ will reduce as $c\tau$ does (since c is closed under τ, it can only use terms in τ' that already were in τ) and thus will normalize. □

Definition 8 (Orthogonality). *Given a pole $\perp\!\!\!\perp$, we say that a term-in-store $(t|\tau)$ is* orthogonal *to a context-in-store $(e|\tau')$ and write $(t|\tau)\perp\!\!\!\perp(e|\tau')$ if τ and τ' are compatible and $\langle t \| e \rangle_{\overline{\tau\tau'}} \in \perp\!\!\!\perp$.*

Remark 9. The reader familiar with Krivine's forcing machine [18] might recognize his definition of orthogonality between terms of the shape (t, p) and stacks of the shape (π, q), where p and q are forcing conditions[8]:

$$(t, p)\perp\!\!\!\perp(\pi, q) \Leftrightarrow (t \star \pi, p \wedge q) \in \perp\!\!\!\perp$$

[8] The meet of forcing conditions is indeed a refinement containing somewhat the "union" of information contained in each, just like the union of two compatible stores.

We can now relate closed terms and contexts by orthogonality with respect to a given pole. This allows us to define for any formula A the sets $|A|_v, |A|_V, |A|_t$ (resp. $\|A\|_F$,$\|A\|_E$, $\|A\|_e$) of realizers (or reducibility candidates) at level v, V, t (resp. F, E, e) for the formula A. It is to be observed that realizers are here closed terms-in-store.

Definition 10 (Realizers). *Given a fixed pole $\perp\!\!\!\perp$, we set:*

$$\begin{aligned}
|X|_v &= \{(\mathbf{k}|\tau) : \ \vdash \mathbf{k} : X\} \\
|A \to B|_v &= \{(\lambda x.t|\tau) : \forall u\tau', \tau \diamond \tau' \wedge (u|\tau') \in |A|_t \Rightarrow (t|\overline{\tau\tau'}[x := u]) \in |B|_t\} \\
\|A\|_F &= \{(F|\tau) : \forall v\tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau') \perp\!\!\!\perp (F|\tau)\} \\
|A|_V &= \{(V|\tau) : \forall F\tau', \tau \diamond \tau' \wedge (F|\tau') \in \|A\|_F \Rightarrow (V|\tau) \perp\!\!\!\perp (F|\tau')\} \\
\|A\|_E &= \{(E|\tau) : \forall V\tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (V|\tau') \perp\!\!\!\perp (E|\tau)\} \\
|A|_t &= \{(t|\tau) : \forall E\tau', \tau \diamond \tau' \wedge (E|\tau') \in \|A\|_E \Rightarrow (t|\tau) \perp\!\!\!\perp (E|\tau')\} \\
\|A\|_e &= \{(e|\tau) : \forall t\tau', \tau \diamond \tau' \wedge (t|\tau') \in |A|_t \Rightarrow (t|\tau') \perp\!\!\!\perp (e|\tau)\}
\end{aligned}$$

Remark 11. We draw the reader attention to the fact that we should actually write $|A|_v^{\perp\!\!\!\perp}$, $\|A\|_F^{\perp\!\!\!\perp}$, etc... and $\tau \Vdash_{\perp\!\!\!\perp} \Gamma$, because the corresponding definitions are parameterized by a pole $\perp\!\!\!\perp$. As it is common in Krivine's classical realizability, we ease the notations by removing the annotation $\perp\!\!\!\perp$ whenever there is no ambiguity on the pole. Besides, it is worth noting that if co-constants do not occur directly in the definitions, they may still appear in the realizers by mean of the pole.

If the definition of the different sets might seem complex at first sight, we claim that they are quite natural in regards of the methodology of Danvy's semantics artifacts presented in [2]. Indeed, having an abstract machine in context-free form (the last step in this methodology before deriving the CPS) allows us to have both the term and the context (in a command) that behave independently of each other. Intuitively, a realizer at a given level is precisely a term which is going to behave well (be in the pole) in front of any opponent chosen in the previous level (in the hierarchy v, F, V,etc...). For instance, in a call-by-value setting, there are only three levels of definition (values, contexts and terms) in the interpretation, because the abstract machine in context-free form also has three. Here the ground level corresponds to strong values, and the other levels are somewhat defined as terms (or context) which are well-behaved in front of any opponent in the previous one. The definition of the different sets $|A|_v, \|A\|_F, |A|_V$, etc... directly stems from this intuition.

In comparison with the usual definition of Krivine's classical realizability, we only considered orthogonal sets restricted to some syntactical subcategories. However, the definition still satisfies the usual monotonicity properties of bi-orthogonal sets:

Proposition 12. *For any type A and any given pole $\perp\!\!\!\perp$, we have:*

1. $|A|_v \subseteq |A|_V \subseteq |A|_t$;
2. $\|A\|_F \subseteq \|A\|_E \subseteq \|A\|_e$.

Proof. All the inclusions are proved in a similar way. We only give the proof for $|A|_v \subseteq |A|_V$. Let $\perp\!\!\!\perp$ be a pole and $(v|\tau)$ be in $|A|_v$. We want to show that $(v|\tau)$

is in $|A|_V$, that is to say that v is in the syntactic category V (which is true), and that for any $(F|\tau') \in \|A\|_F$ such that $\tau \diamond \tau'$, $(v|\tau)\perp\!\!\!\perp(F|\tau')$. The latter holds by definition of $(F|\tau') \in \|A\|_F$, since $(v|\tau) \in |A|_v$. □

We now extend the notion of realizers to stores, by stating that a store τ realizes a context Γ if it binds all the variables x and α in Γ to a realizer of the corresponding formula.

Definition 13. *Given a closed store τ and a fixed pole $\perp\!\!\!\perp$, we say that τ* realizes *Γ, which we write*[9] *$\tau \Vdash \Gamma$, if:*

1. *for any $(x : A) \in \Gamma$, $\tau \equiv \tau_0[x := t]\tau_1$ and $(t|\tau_0) \in |A|_t$*
2. *for any $(\alpha : A^\perp) \in \Gamma$, $\tau \equiv \tau_0[\alpha := E]\tau_1$ and $(E|\tau_0) \in \|A\|_E$*

In the same way than weakening rules (for the typing context) are admissible for each level of the typing system :

$$\frac{\Gamma \vdash_t t : A \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_t t : A} \qquad \frac{\Gamma \vdash_e e : A^\perp \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_e e : A^\perp} \qquad \cdots \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma'' \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_\tau \tau : \Gamma''}$$

the definition of realizers is compatible with a weakening of the store.

Lemma 14 (Store weakening). *Let τ and τ' be two stores such that $\tau \lhd \tau'$, let Γ be a typing context and let $\perp\!\!\!\perp$ be a pole. The following statements hold:*

1. *$\overline{\tau\tau'} = \tau'$*
2. *If $(t|\tau) \in |A|_t$ for some closed term $(t|\tau)$ and type A, then $(t|\tau') \in |A|_t$. The same holds for each level e, E, V, F, v of the typing rules.*
3. *If $\tau \Vdash \Gamma$ then $\tau' \Vdash \Gamma$.*

Proof. 1. Straightforward from the definition of $\tau\bar{\tau}'$.

2. This essentially amounts to the following observations. First, one remarks that if $(t|\tau)$ is a closed term, so then so is $(t|\overline{\tau\tau'})$ for any closed store τ' compatible with τ. Second, we observe that if we consider for instance a closed context $(E|\tau'') \in \|A\|_E$, then $\overline{\tau\tau'} \diamond \tau''$ implies $\tau \diamond \tau''$, thus $(t|\tau)\perp\!\!\!\perp(E|\tau'')$ and finally $(t|\overline{\tau\tau'})\perp\!\!\!\perp(E|\tau'')$ by closure of the pole under store extension. We conclude that $(t|\tau')\perp\!\!\!\perp(E|\tau'')$ using the first statement.
3. By definition, for all $(x : A) \in \Gamma$, τ is of the form $\tau_0[x := t]\tau_1$ such that $(t|\tau_0) \in |A|_t$. As τ and τ' are compatible, we know by Lemma 4 that $\overline{\tau\tau'}$ is of the form $\tau_0'[x := t]\tau_1'$ with τ_0' an extension of τ_0, and using the first point we get that $(t|\tau_0') \in |A|_t$. □

Definition 15 (Adequacy). *Given a fixed pole $\perp\!\!\!\perp$, we say that:*

– *A typing judgment $\Gamma \vdash_t t : A$ is* adequate *(w.r.t. the pole $\perp\!\!\!\perp$) if for all stores $\tau \Vdash \Gamma$, we have $(t|\tau) \in |A|_t$.*

[9] Once again, we should formally write $\tau \Vdash_{\perp\!\!\!\perp} \Gamma$ but we will omit the annotation by $\perp\!\!\!\perp$ as often as possible.

– *More generally, we say that an inference rule*

$$\frac{J_1 \quad \cdots \quad J_n}{J_0}$$

is adequate (w.r.t. the pole $\perp\!\!\!\perp$) if the adequacy of all typing judgments $J_1, \ldots, J_n$ implies the adequacy of the typing judgment J_0.

Remark 16. From the latter definition, it is clear that a typing judgment that is derivable from a set of adequate inference rules is adequate too.

We will now show the main result of this section, namely that the typing rules of Figure 2 for the $\overline{\lambda}_{[lv\tau\star]}$-calculus without co-constants are adequate with any pole. Observe that this result requires to consider the $\overline{\lambda}_{[lv\tau\star]}$-calculus without co-constants. Indeed, we consider co-constants as coming with their typing rules, potentially giving them any type (whereas constants can only be given an atomic type). Thus, there is *a priori* no reason[10] why their types should be adequate with any pole.

However, as observed in the previous remark, given a fixed pole it suffices to check whether the typing rules for a given co-constant are adequate with this pole. If they are, any judgment that is derivable using these rules will be adequate.

Theorem 17 (Adequacy). *If Γ is a typing context, $\perp\!\!\!\perp$ is a pole and τ is a store such that $\tau \Vdash \Gamma$, then the following holds in the $\overline{\lambda}_{[lv\tau\star]}$-calculus without co-constants:*

1. *If v is a strong value such that $\Gamma \vdash_v v : A$, then $(v|\tau) \in |A|_v$.*
2. *If F is a forcing context such that $\Gamma \vdash_F F : A^\perp$, then $(F|\tau) \in \|A\|_F$.*
3. *If V is a weak value such that $\Gamma \vdash_V V : A$, then $(V|\tau) \in |A|_V$.*
4. *If E is a catchable context such that $\Gamma \vdash_E E : A^\perp$, then $(E|\tau) \in \|A\|_F$.*
5. *If t is a term such that $\Gamma \vdash_t t : A$, then $(t|\tau) \in |A|_t$.*
6. *If e is a context such that $\Gamma \vdash_e e : A^\perp$, then $(e|\tau) \in \|A\|_e$.*
7. *If c is a command such that $\Gamma \vdash_c c$, then $c\tau \in \perp\!\!\!\perp$.*
8. *If τ' is a store such that $\Gamma \vdash_\tau \tau' : \Gamma'$, then $\tau\tau' \Vdash \Gamma, \Gamma'$.*

Proof. The different statements are proved by mutual induction over typing derivations. We only give the most important cases here.

Rule $(\to_l)$. Assume that

$$\frac{\Gamma \vdash_t u : A \quad \Gamma \vdash_E E : B^\perp}{\Gamma \vdash_F u \cdot E : (A \to B)^\perp} \ (\to_l)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(\lambda x.t|\tau')$ be a closed term in the set $|A \to B|_v$ such that $\tau \diamond \tau'$, then we have:

$$\langle \lambda x.t \| u \cdot E \rangle \overline{\tau\tau'} \quad \to \quad \langle u \| \tilde{\mu} x.\langle t \| E \rangle \rangle \overline{\tau\tau'} \quad \to \quad \langle t \| E \rangle \overline{\tau\tau'}[x := u]$$

By definition of $|A \to B|_v$, this closure is in the pole, and we can conclude by anti-reduction.

[10] Think for instance of a co-constant of type $(A \to B)^\perp$, there is no reason why it should be orthogonal to any function in $|A \to B|_v$.

Rule (x). Assume that

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_V x : A} \ (x)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. As $(x : A) \in \Gamma$, we know that τ is of the form $\tau_0[x := t]\tau_1$ with $(t|\tau_0) \in |A|_t$. Let $(F|\tau')$ be in $\|A\|_F$, with $\tau \diamond \tau'$. By Lemma 4, we know that $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x := t]\overline{\tau_1}$. Hence we have:

$$\langle x \| F \rangle \overline{\tau_0}[x := t]\overline{\tau_1} \quad \to \quad \langle t \| \tilde{\mu}[x].\langle x \| F \rangle \overline{\tau_1} \rangle \overline{\tau_0}$$

and it suffices by anti-reduction to show that the last closure is in the pole $\perp\!\!\!\perp$. By induction hypothesis, we know that $(t|\tau_0) \in |A|_t$ thus we only need to show that it is in front of a catchable context in $\|A\|_E$. This corresponds exactly to the next case that we shall prove now.

Rule $(\tilde{\mu}^{[]})$. Assume that

$$\frac{\Gamma, x : A, \Gamma' \vdash_F F : A \quad \Gamma, x : A \vdash \tau' : \Gamma'}{\Gamma \vdash_E \tilde{\mu}[x].\langle x \| F \rangle \tau' : A} \ (\tilde{\mu}^{[]})$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(V|\tau_0)$ be a closed term in $|A|_V$ such that $\tau_0 \diamond \tau$. We have that :

$$\langle V \| \tilde{\mu}[x].\langle x \| F \rangle \overline{\tau'} \rangle \overline{\tau_0 \tau} \quad \to \quad \langle V \| F \rangle \overline{\tau_0 \tau}[x := V]\tau'$$

By induction hypothesis, we obtain $\tau[x := V]\tau' \Vdash \Gamma, x : A, \Gamma'$. Up to α-conversion in F and τ', so that the variables in τ' are disjoint from those in τ_0, we have that $\overline{\tau_0\tau} \Vdash \Gamma$ (by Lemma 14) and then $\tau'' \triangleq \overline{\tau_0\tau}[x := V]\tau' \Vdash \Gamma, x : A, \Gamma'$. By induction hypothesis again, we obtain that $(F|\tau'') \in \|A\|_F$ (this was an assumption in the previous case) and as $(V|\tau_0) \in |A|_V$, we finally get that $(V|\tau_0) \perp\!\!\!\perp (F|\tau'')$ and conclude again by anti-reduction. □

Corollary 18. *If $c\tau$ is a closure such that $\vdash_l c\tau$ is derivable, then for any pole $\perp\!\!\!\perp$ such that the typing rules for co-constants used in the derivation are adequate with $\perp\!\!\!\perp$, $c\tau \in \perp\!\!\!\perp$.*

We can now put our focus back on the normalization of typed closures. As we already saw in Proposition 7, the set $\perp\!\!\!\perp_{\Downarrow}$ of normalizing closure is a valid pole, so that it only remains to prove that any typing rule for co-constants is adequate with $\perp\!\!\!\perp_{\Downarrow}$.

Lemma 19. *Any typing rule for co-constants is adequate with the pole $\perp\!\!\!\perp_{\Downarrow}$,* i.e. *if Γ is a typing context, and τ is a store such that $\tau \Vdash \Gamma$, if $\boldsymbol{\kappa}$ is a co-constant such that $\Gamma \vdash_F \boldsymbol{\kappa} : A^{\perp}$, then $(\boldsymbol{\kappa}|\tau) \in \|A\|_F$.*

Proof. This lemma directly stems from the observation that for any store τ and any closed strong value $(v|\tau') \in |A|_v$, $\langle v \| \boldsymbol{\kappa} \rangle \overline{\tau\tau'}$ does not reduce and thus belongs to the pole $\perp\!\!\!\perp_{\Downarrow}$.

As a consequence, we obtain the normalization of typed closures of the full calculus.

Theorem 20. *If $c\tau$ is a closure of the $\overline{\lambda}_{[lv\tau\star]}$-calculus such that $\vdash_l c\tau$ is derivable, then $c\tau$ normalizes.*

This is to be contrasted with Okasaki, Lee and Tarditi's semantics for the call-by-need λ-calculus, which is not normalizing in the simply-typed case, as shown in Ariola *et al.* [2].

2.3 Extension to 2nd-order type systems

We focused in this article on simply-typed versions of the $\overline{\lambda}_{lv}$ and $\overline{\lambda}_{[lv\tau\star]}$ calculi. But as it is common in Krivine classical realizability, first and second-order quantifications (in Curry style) come for free through the interpretation. This means that we can for instance extend the language of types to first and second-order predicate logic:

$$\begin{array}{l} e_1, e_2 ::= x \mid f(e_1, \ldots, e_k) \\ A, B ::= X(e_1, \ldots, e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \end{array}$$

We can then define the following introduction rules for universal quantifications:

$$\frac{\Gamma \vdash_v v : A \quad x \notin FV(\Gamma)}{\Gamma \vdash_v v : \forall x.A}\ (\forall^1_r) \qquad \frac{\Gamma \vdash_v v : A \quad X \notin FV(\Gamma)}{\Gamma \vdash_v v : \forall X.A}\ (\forall^2_r)$$

Observe that these rules need to be restricted at the level of strong values, just as they are restricted to values in the case of call-by-value[11]. As for the left rules, they can be defined at any levels, let say the more general e:

$$\frac{\Gamma \vdash_e e : (A[n/x])^{\perp\!\!\!\perp}}{\Gamma \vdash_e e : (\forall x.A)^{\perp\!\!\!\perp}}\ (\forall^1_l) \qquad \frac{\Gamma \vdash_e e : (A[B/X])^{\perp\!\!\!\perp}}{\Gamma \vdash_e e : (\forall X.A)^{\perp\!\!\!\perp}}\ (\forall^2_l)$$

where n is any natural number and B any formula. The usual (call-by-value) interpretation of the quantification is defined as an intersection over all the possible instantiations of the variables within the model. We do not wish to enter into too many details[12] on this topic here, but first-order variable are to be instantiated by integers, while second order are to be instantiated by subset of terms at the lower level, *i.e.* closed strong-values in store (which we write $\mathcal{V}_0$):

$$|\forall x.A|_v = \bigcap_{n \in \mathbb{N}} |A[n/x]|_v \qquad |\forall X.A|_v = \bigcap_{S \in \mathbb{N}^k \to \mathcal{P}(\mathcal{V}_0)} |A[S/X]|_v$$

where the variable X is of arity k. It is then routine to check that the typing rules are adequate with the realizability interpretation.

[11] For further explanation on the need for a value restriction in Krivine realizability, we refer the reader to [29] or [25].

[12] Once again, we advise the interested reader to refer to [29] or [25] for further details.

3 Conclusion and further work

In this paper, we presented a system of simple types for a call-by-need calculus with control, which we proved to be safe in that it satisfies subject reduction (Theorem 1) and that typed terms are normalizing (Theorem 20). We proved the normalization by means of realizability-inspired interpretation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus. Incidentally, this opens the doors to the computational analysis (in the spirit of Krivine realizability) of classical proofs using control, laziness and shared memory.

In further work, we intend to present two extensions of the present paper. First, following the definition of the realizability interpretation, we managed to type the continuation-and-store passing style translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus (see [2]). Interestingly, typing the translation emphasizes its computational content, and in particular, the store-passing part is reflected in a Kripke forcing-like manner of typing the extensibility of the store [28, Chapter 6].

Second, on a different aspect, the realizability interpretation we introduced could be a first step towards new ways of realizing axioms. In particular, the first author used in his Ph.D. thesis [28, Chapter 8] the techniques presented in this paper to give a normalization proof for dPA^{ω}, a proof system developed by the second author [15]. Indeed, this proof system allows to define a proof for the axiom of dependent choice thanks to the use of streams that are lazily evaluated, and was lacking a proper normalization proof.

Finally, to determine the range of our technique, it would be natural to investigate the relation between our framework and the many different presentations of call-by-need calculi (with or without control). Amongst other calculi, we could cite Chang-Felleisen presentation of call-by-need [4], Garcia *et al.* lazy calculus with delimited control [10] or Kesner's recent paper on normalizing by-need terms characterized by an intersection type system [16]. To this end, we might rely on Pédrot and Saurin's classical by-need [33]. They indeed relate (classical) call-by-need with linear head-reduction from a computational point of view, and draw the connections with the presentations of Ariola *et al.* [2] and Chang-Felleisen [4]. Ariola *et al.* $\overline{\lambda}_{lv}$-calculus being close to the $\overline{\lambda}_{[lv\tau\star]}$-calculus (see [2] for further details), our technique is likely to be adaptable to their framework, and thus to Pédrot and Saurin's system.

References

1. Zena Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1993.
2. Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In Tom Schrijvers and Peter Thiemann, editors, *Proceedings of FLOPS'12, Kobe, Japan, May 23-25, 2012. Proceedings*, LNCS, pages 32–46. Springer, 2012.
3. Franco Barbanera and Stefano Berardi. A symmetric λ-calculus for classical program extraction. *Information and Computation*, 125(2):103–117, 1996.

4. Stephen Chang and Matthias Felleisen. The call-by-need lambda calculus, revisited. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 128–147, 2012.
5. Tristan Crolard. A confluent lambda-calculus with a catch/throw mechanism. *J. Funct. Program.*, 9(6):625–647, 1999.
6. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of ICFP 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000.
7. Pierre-Évariste Dagand and Gabriel Scherer. Normalization by realizability also evaluates. In David Baelde and Jade Alglave, editors, *Proceedings of JFLA'15*, Le Val d'Ajol, France, January 2015.
8. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. Reasoning with continuations. In *Proceedings of LICS'86, Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 131–141. IEEE Computer Society, 1986.
9. Jean Gallier. On girard's 'candidats de reductibilité.'. In Odifreddi, editor, *Logic and Computer Science*, pages 123–203. Academic Press, 1900.
10. Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. *Logical Methods in Computer Science*, Volume 6, Issue 3, July 2010.
11. Jean-Yves Girard. Une extension de Ľinterpretation de gödel a Ľanalyse, et son application a Ľelimination des coupures dans Ľanalyse et la theorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63 – 92. Elsevier, 1971.
12. Mauricio Guillermo and Alexandre Miquel. Specifying peirce's law in classical realizability. *Mathematical Structures in Computer Science*, 26(7):1269–1303, 2016.
13. Mauricio Guillermo and Étienne Miquey. Classical realizability and arithmetical formulæ. *Mathematical Structures in Computer Science*, page 1–40, 2016.
14. Hugo Herbelin. *C'est maintenant qu'on calcule: au cœur de la dualité*. Habilitation thesis, University Paris 11, December 2005.
15. Hugo Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 365–374. IEEE Computer Society, 2012.
16. Delia Kesner. *Reasoning About Call-by-need by Means of Types*, pages 424–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
17. J.-L. Krivine. Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.
18. J.-L. Krivine. Realizability algebras: a program to well order r. *Logical Methods in Computer Science*, 7(3), 2011.
19. J.-L. Krivine. Realizability algebras II : new models of ZF + DC. *Logical Methods in Computer Science*, 8(1):10, February 2012. 28 p.
20. J.-L. Krivine. Quelques propriétés des modèles de réalisabilité de ZF, February 2014.
21. Jean-Louis Krivine. *Lambda-calculus, types and models*. Ellis Horwood series in computers and their applications. Masson, 1993.
22. Jean-Louis Krivine. Realizability in classical logic. In interactive models of computation and program behaviour. *Panoramas et synthèses*, 27, 2009.
23. Yves Lafont, Bernhard Reus, and Thomas Streicher. Continuations semantics or expressing implication by negation. Technical Report 9321, Ludwig-Maximilians-Universität, München, 1993.

24. Frédéric Lang. Explaining the lazy krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation*, 20(3):257–270, Sep 2007.
25. Rodolphe Lepigre. A classical realizability model for a semantical value restriction. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.
26. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.
27. Alexandre Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods in Computer Science*, 7(2):188–202, 2011.
28. Étienne Miquey. *Classical realizability and side-effects*. PhD thesis, Université Paris-Diderot, Universidad de la República (Uruguay), 2017.
29. Guillaume Munch-Maccagnoni. Focalisation and Classical Realisability. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic '09*, volume 5771 of *Lecture Notes in Computer Science*, pages 409–423. Springer, Heidelberg, 2009.
30. Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, 1994.
31. Michel Parigot. Free deduction: An analysis of "computations" in classical logic. In Andrei Voronkov, editor, *Proceedings of LPAR*, volume 592 of *LNCS*, pages 361–380. Springer, 1991.
32. Michel Parigot. Strong normalization of second order symmetric lambda-calculus. In Sanjiv Kapoor and Sanjiva Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science, 20th Conference, FST TCS 2000 New Delhi, India, December 13-15, 2000, Proceedings*, volume 1974 of *LNCS*, pages 442–453. Springer, 2000.
33. Pierre-Marie Pédrot and Alexis Saurin. *Classical By-Need*, pages 616–643. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
34. Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
35. Emmanuel Polonowski. Strong normalization of lambda-mu-mu/tilde-calculus with explicit substitutions. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2987 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2004.
36. W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.