# Continuation-and-environment-passing style translations: a focus on call-by-need

Hugo Herbelin, Étienne Miquey

# Continuation-and-environment-passing style translations: a focus on call-by-need

Hugo Herbelin[1] and Étienne Miquey[2]

[1] Équipe $\pi r^2$, INRIA, IRIF, Université herbelin@inria
[2] Équipe Gallinette, INRIA, LS2N, Université de Nantes
etienne.miquey@inria.fr

**Abstract.** The call-by-need evaluation strategy for the $\lambda$-calculus is an evaluation strategy that lazily evaluates arguments only if needed, and if so, shares computations across all places where it is needed. To implement this evaluation strategy, abstract machines require some form of global environment. While abstract machines usually lead to a better understanding of the flow of control during the execution, easing in particular the definition of continuation-passing style translations, the case of machines with global environments turns out to be much more subtle. The main purpose of this paper is to understand how to type a continuation-and-environment-passing style translations, that it to say how to soundly translate a classical calculus with environment into a calculus that does not have these features. To this end, we focus on a sequent calculus presentation of a call-by-need $\lambda$-calculus with classical control for which Ariola *et. al* already defined an untyped translation [5] and which we equipped with a system of simple types in a previous paper [32]. We present here a type system for the target language of their translation, which highlights a variant of Kripke forcing related to the environment-passing part of the translation. Finally, we show that our construction naturally handles the cases of call-by-name and call-by-value calculi with environment, encompassing in particular the Milner Abstract Machine, a machine with global environments for the call-by-name $\lambda$-calculus.

## 1 Introduction

### 1.1 Continuation-passing style translations

The terminology of *continuation-passing style* (CPS) was first introduced in 1975 by Sussman and Steele in a technical report about the Scheme programming language [40]. In this report, after giving the usual recursive definition of the factorial, they explained how the same computation could be driven differently:

> *"It is always possible, if we are willing to specify explicitly what to do with the answer, to perform any calculation in this way: rather than reducing to its value, it reduces to an application of a continuation to its value. That is, in this continuation-passing programming style, a function always "returns" its result by "sending" it to another function. This is the key idea."*

Interestingly, by making explicit the order in which reduction steps are computed, continuation-passing style translations indirectly specify an operational semantics for the translated calculus. In particular, different evaluation strategies for a calculus correspond to different continuation-passing style translations.

Continuations and their computational benefits have been deeply studied since then, and there exists a wide literature on continuation-passing style translations. Among other things, these translations have been used to ease the definitions of compilers [3,14], one of their interests being that they make explicit the flow of control. As such, continuation-passing style translations *de facto* provide us with an operational semantics for control operators, as observed in [15] for the $\mathcal{C}$ operator.

It is well-known since the seminal paper of Griffin that in the realm of the proofs-as-programs correspondence, control operators gives a tangible computational content to classical logic [16]. Continuation-passing style translations therefore bring an indirect computational interpretation of classical logic. This observation can be strengthened on a purely logical aspect by considering the logical translations they induce at the level of types: the translation of types through a CPS mostly amounts to a negative translation allowing to embed classical logic into intuitionistic logic [16,34].

### 1.2 Call-by-need

Call-by-need evaluation strategy of the $\lambda$-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 $\lambda$-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 semantics characterizing these theories [38]. 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 [4], and by Maraist, Odersky and Wadler [28]. A continuation-passing style semantics was proposed in the 90s by Okasaki, Lee and Tarditi [35]. However, this semantics does not ensure normalization of simply-typed terms, as shown in [5], thus failing to ensure a property which holds in the simply-typed call-by-name and call-by-value cases.

### 1.3 CPS, abstract machines and sequent calculi

The main difficulty in deriving a CPS translation for a call-by-need calculus is related to the necessity of sharing computations, a problem which can already be observed when trying to define an abstract machine. While most of the abstract machines for the $\lambda$-calculus evaluate terms within a local environment by using closures (*e.g.* Landin's SECD machine [24], the Krivine Abstract Machine [22], *etc.*), this is not enough for call-by-need calculi. Instead, the call-by-need evaluation strategy requires abstract machines with a global environment[3]. Such machines demand in particular to explicitly handle addresses (as in the Lazy Krivine [25]) or a renaming process (as in the Milner Abstract Machine [1]).

It is folklore than continuation-passing style translations for different flavors of $\lambda$-calculi can be advantageously factorized through sequent calculi. The semantics of calculi with control can indeed be reconstructed from an analysis of the duality between programs and their evaluation contexts. Such an analysis can be performed in the context of sequent calculi such as Curien-Herbelin's $\bar{\lambda}\mu\tilde{\mu}$-calculus, in which evaluation contexts (and thus continuations) are reified into concrete object of the syntax [11]. As such, the operational semantics of sequent calculi are thus really close from the one of abstract machines.

Defining a sequent calculus with the adequate operational semantics can thus constitute a good intermediate step in order to ease the further definition of a continuation-passing style translation. Based on this idea, Danvy proposed a methodology to mechanically obtain CPS translations from the successive derivations of several semantics artifacts [12]. This methodology is applied in [5] to get a continuation-passing style translation for a call-by-need calculus with control using several call-by-need sequent calculi. The $\overline{\lambda}_{[lv\tau\star]}$-calculus, an abstract machine-like sequent calculus they defined, as well as the untyped CPS translation they obtained, are the main objects of study of this paper.

### 1.4 A typed continuation-and-environment-passing style translation

The main purpose of this paper will be to type the CPS translation from [5], that is to say to define a type system suitable to be the target calculus of the translation. In [5], the translation is indeed untyped, therefore failing at proving the normalization or the soundness of the $\overline{\lambda}_{[lv\tau\star]}$-calculus. In a recent paper [32], these properties have been proved by means of a realizability interpretation *à la* Krivine. The structure of this interpretation turns out to be very instructive on the difficulties encountered when trying to type the CPS translation.

First, since the evaluation of terms is shared, the continuation-passing style translation has actually to be combined with an environment-passing style transformation. Second, as environments can grow during the execution, the realizability interpretation is defined to be compatible with environments extension[4].

[3] For a discussion on the benefits and drawbacks of local and global environment, we refer the reader to [2].

[4] Namely, the definitions of pole and truth/falsity values all include a component made of an environment, and are stable under environment extension.

The type translation will internalize this extensibility by means of a variant of Kripke forcing. Last but not least, the translation has to handle problems related to the uniqueness of names. In a nutshell, this is due to the fact that terms can contain unbound variables that refer to elements of the environment. Therefore, a collision of names can result in self-references and non-terminating terms. We deal with this by refining the translation to use de Bruijn levels to access elements of the environment, which also have the advantage of making it closer to an actual implementation. Surprisingly, the passage to de Bruijn levels also unveils some computational content related to the extension of environments.

The target calculus of our translation will then include typed stores and a bounded second-order quantification over stores types, written $\forall X <: \Upsilon$ and reading "*for any* $X$ *extending* $\Upsilon$", in order to account for store extensions. This quantification is reminiscent of Cardelli's System $F_{<:}$ [9], and we will name our calculus $F_\Upsilon$. Also, since we will use de Bruijn levels to access the elements of a store (*i.e.* the variable $n$ refers to the $n^{\text{th}}$ cell of the store), an arbitrary extension of the store may result in the necessity of updating de Bruijn levels to take this extension into account. To this purpose, we will use explicit coercions to witness the extension of stores. Interestingly, we will see that the definition of System $F_\Upsilon$ and the structure of the translation are related to the use of global environments but that they are not peculiar to the call-by-need evaluation strategy.

### 1.5 Organization of the paper

We briefly outline the organization of the paper.

- Section 2 is devoted to a detailed introduction of the sequent calculi and abstract machines using global environments that we study in this paper. We begin with the introduction of the Milner Abstract Machine (MAM) for the call-by-name $\lambda$-calculus in Section 2.1, which we relate in Section 2.2 to a variant of the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments. We then present the by-need version of the MAM (Section 2.3), and we introduce a variant with de Bruijn levels of Ariola *et al.*'s sequent calculus presentation of a call-by-need $\lambda$-calculus with control, called the $\overline{\lambda}_{[lv\tau\star]}$-calculus (Section 2.4).
- Section 3 focuses on the target system of the typed continuation-and-environment-passing style translations that we present in the sequel. We first highlight the guidelines of the translations in Section 3.1 and in particular the Kripke forcing-like manner of anticipating environments extensions, before introducing System $F_\Upsilon$ in Section 3.2 and some of its properties in Section 3.3.
- Section 4 dwells on two specific continuations-and- environments-passing style translations: a call-by-need translation in Section 4.1 and a call-by-name translation in Section 4.2. Both translations are shown to be type-preserving thanks to the properties of System $F_\Upsilon$.
- Finally, we conclude in Section 5 by emphasizing the connection between environment and Kripke forcing, before presenting perspectives and further work.

$$
\begin{array}{rcl|rcl}
t,u & ::= & x \mid tu \mid \lambda x.t & \overline{t}\overline{u} \star \pi \star \tau & \to_{\mathtt{c}} & \overline{t} \star \overline{u} \cdot \pi \star \tau \\
\pi & ::= & \varepsilon \mid \overline{t} \cdot \pi & \lambda x.\overline{t} \star \overline{u} \cdot \pi \star \tau & \to_{\beta} & \overline{t} \star \pi \star \tau[x := \overline{u}] \\
\tau & ::= & \varepsilon \mid \tau[x := t] & x \star \pi \star \tau[x := \overline{t}]\tau' & \to_{\mathtt{s}} & \overline{t}^{\alpha} \star \pi \star \tau[x := \overline{t}]\tau'
\end{array}
$$

(a) Milner Abstract Machine

$$
\begin{array}{rcl|rcl}
t,u & ::= & x \mid tu \mid \lambda x.t & \overline{t}\overline{u} \star S \star E & \to_{\mathtt{c}} & \overline{t} \star (\overline{u}, E) \cdot S \star E \\
S & ::= & \varepsilon \mid (t, E) \cdot S & \lambda x.\overline{t} \star (\overline{u}, E') \cdot S \star E & \to_{\beta} & \overline{t} \star S \star E[x ::= (\overline{u}, E')] \\
E & ::= & \varepsilon \mid E[x ::= (t, E')] & x \star S \star E[x ::= (\overline{t}, E')]E'' & \to_{\mathtt{s}} & \overline{t} \star S \star E'
\end{array}
$$

(b) Krivine Abstract Machine

**Fig. 1.** MAM vs KAM

## 2 Computing with global environments

In this section, we recall and introduce several calculi and abstract machines that have in common that they use a form of global environments to perform substitutions. As such, what we call global environments somewhat behave like (lazy) explicit substitutions or particular stores. To draw the comparison with 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 its cells can be updated at any time and thus values might be replaced. In our case, cells might be updated but only to replace an unevaluated term by its corresponding value. Second, the usual notion of environment designates a structure in which variables are bound 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 resembles 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.

### 2.1 The Milner Abstract Machine

Even though it is far from being the most known abstract machine, the Milner Abstract Machine (MAM) is probably the easiest presentation of an abstract machine for the (call-by-name) $\lambda$-calculus that uses a single global environment [1,2]. A state of this machine is made of three components:

- a code $\overline{t}$ for a term $t$ which is not considered up to $\alpha$-conversion,
- a stack $\pi$ which contains the arguments of the current code, that is to say a stack of codes,
- a global environment $\tau$, which is a list storing the (delayed) substitutions generated by the redexes encountered so far.

The machine is given in Figure 1, where we follow the notations of [2] to denote reduction rules: we write $\to_{\beta}$ for the $\beta$-reduction rule, $\to_{\mathtt{s}}$ for the rule which

somehow performs a *substitution*, and $\to_{\mathtt{c}}$ for the *commutative* transition. By considering invariants of the machine, one can easily prove that the Milner Abstract Machine soundly implements the call-by-name evaluation strategy for the $\lambda$-calculus [1].

It is worth noting that the soundness of the execution crucially relies on the uniqueness of names in the environment. Incidentally, this requires the possibility of an on-the-fly $\alpha$-renaming process. Executing twice a term bounding a variable $x$ in different contexts (say $(\lambda x.t)u$ and $(\lambda x.t)v$) could indeed result in linking two different terms to the same name in the environment. This is avoided by asking in the $\to_{\mathtt{s}}$ rule that if $x$ is linked to some term $t$ in the environment $\tau$, accessing $x$ results in executing a code $\overline{t}^{\alpha}$ that is $\alpha$-equivalent to $t$ and such that any bound name in $\overline{t}^{\alpha}$ is fresh with respect to those in the current stack $\pi$ and environment $\tau$. In the sequel, we will explicitly use De Bruijn levels to handle this kind of issues.

On the contrary, most of the abstract machines of the literature implementing the call-by-name or call-by-value evaluation strategies for the $\lambda$-calculus (*e.g.* the Krivine Abstract Machine [22], Landin's SECD machine [24], Felleisen and Friedman's CEK machine [13], Leroy's ZINC machine [26]) uses many local environments. In these machines, the concept of *closure*, that is a term taken with an environment under which it can be seen as a closed term, plays a central part. As an example, we give in Figure 1 the definition of the Krivine Abstract Machine (KAM). In comparison with the MAM, it is interesting to observe that the definitions of environments and closures are mutually recursive. Notably, it presents the advantage that the locality of environments makes the $\alpha$-renaming process useless. While this design with a local environment presents some benefits over machines with global ones (among other things in terms of complexity [2]), it has the drawback of being incompatible with lazy evaluation strategies which require to share computations and memory bindings.

### 2.2 The $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments

Before dwelling on the by-need variant of the MAM, let us briefly explain how the MAM could easily be expressed under the shape of a sequent calculus. We present here a variant of the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus extended with a global environment [11]. The syntax of the usual $\bar{\lambda}\mu\tilde{\mu}$-calculus is divided in three categories: *terms*, which represent programs; *evaluation contexts* (or co-terms); *commands*, which are pairs of a term and a context representing a system that contains both the program and its environment. Then, as in the MAM, we extend the syntax with *environments* made of delayed substitutions. The notion of evaluation context is a generalization of the notion of stacks where $\tilde{\mu}a.c$ can be read as a context $\mathtt{let}\, a = [\,]\, \mathtt{in}\, c$. As for terms, the $\mu$ operator comes from Parigot's $\lambda\mu$-calculus [36]: $\mu\alpha$ binds a context to a context variable $\alpha$ in the same way $\tilde{\mu}a$ binds a proof to some proof variable $a$. In particular, as we shall see now, it allows terms to capture evaluation contexts and as such plays the role of a control operator.

| | | | |
|---|---|---|---|
| **Values** | $V ::= \lambda x.t$ | **Environment** | $\tau ::= \varepsilon \mid \tau[x := t] \mid \tau[\alpha := E]$ |
| **Terms** | $t, u ::= V \mid x \mid \mu\alpha.c$ | **Commands** | $c ::= \langle t \parallel e \rangle$ |
| **Co-values** | $E ::= t \cdot E \mid \alpha$ | **Closures** | $l ::= c\tau$ |
| **Contexts** | $e ::= E \mid \tilde{\mu}x.c$ | | |

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

**Fig. 2.** Call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments

The syntax and reduction rules are given in Figure 2, in which terms and contexts are implicitly considered up to $\alpha$-conversion in order to preserve the uniqueness of names in the environment. It is easy to see that on its intuitionistic fragment (that is without the classical control $\mu\alpha$), this calculus behaves exactly as the MAM[5]. Observe that the reduction rules for $\mu$ is restricted to co-values, which enforces the call-by-name reduction strategy. Call-by-value is obtained by relaxing this constraint and by dually constraining the reduction of $\tilde{\mu}$ to values [11]. As for the typing rules, they will be easy to deduce from the type system we will introduce for the $\overline{\lambda}_{[lv\tau\star]}$-calculus in Section 2.4, we thus let them as an exercise for the reader.

### 2.3 The Milner Abstract Machine By-Need

The call-by-need evaluation strategy of the $\lambda$-calculus evaluates arguments of functions only when needed, and, when needed, shares their evaluations across all places where the argument is required. Therefore, abstract machines implementing the call-by-need evaluation have to allow for some kind of global environment in order to share computations [39,10]. The Milner Abstract Machine can easily be modified to obtain such an abstract machine, called the *Milner Abstract machine by-neeD* (MAD) [1]. The main idea consists in adding a *dump*, which is used whenever the code is some variable $x$ within an environment $\tau_1[x := t]\tau_2$: the machine momentarily focuses on the evaluation of $t$ in $\tau_1$ while saving the current stack together with the rest of the environment $\tau_2$ and the variable $x$

[5] The reader unfamiliar with the $\bar{\lambda}\mu\tilde{\mu}$-calculus might be puzzled by the absence of a syntactic construction for the application of proof terms. Intuitively, the usual application $tu$ of the $\lambda$-calculus is replaced by the application of the proof $t$ to a stack of the shape $u \cdot E$. The usual application can be recovered through the following shorthand: $tu \triangleq \mu\alpha.\langle r \parallel u \cdot \alpha \rangle$. It is then an easy exercise to show that the MAM can be simulated within the $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments, see Appendix A.

$$
\begin{array}{llllllll}
\overline{t}\overline{u} & \star & \pi & \star & \tau & \star\ D & \to_{\mathtt{c}_1} & \overline{t} \star \overline{u}\cdot\pi \star \tau \star D \\
\lambda x.\overline{t} & \star & \overline{u}\cdot\pi & \star & \tau & \star\ D & \to_{\beta} & \overline{t} \star \pi \star \tau[x := \overline{u}] \star D \\
x & \star & \pi & \star & \tau[x := \overline{t}]\tau' & \star\ D & \to_{\mathtt{c}_2} & \overline{t} \star \varepsilon \star \tau \star (x, \pi, \tau') :: D \\
\overline{v} & \star & \varepsilon & \star & \tau & \star\ (x, \pi, \tau') :: D & \to_{\mathtt{s}} & \overline{v}^{\alpha} \star \pi \star \tau[x := \overline{v}]\tau' \star D
\end{array}
$$

**Fig. 3.** The Milner Abstract by-neeD machine

on the dump. Then, if this computation eventually produces a value $V$ in an environment $\tau_1'$, the machine goes back to the former computation within the updated environment $\tau_1'[x := V]\tau_2$. The machine is given Figure 3, where environments $\tau$ and stacks $\pi$ are defined as in the MAM, and where dumps are given by the following grammar:

$$D ::= \varepsilon \mid (x, \pi, \tau) :: D$$

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

If the MAD arguably provides us with the easiest presentation of a lazy abstract machine, it does not directly lead to an operational semantics for control operators (or, equivalently, to the definition of a continuation-passing style translation). While the addition to the KAM of the `call/cc` operator, which allows to capture the current stack into a continuation, is very natural, it is less obvious to determine its behavior in the MAD. Especially, it is not clear *a priori* how control operators should handle the global environment and the dump. More generally, the problem of soundly defining a CPS translation for the call-by-need $\lambda$-calculus turns out to be trickier than in the call-by-value and call-by-name cases. In particular, a first attempt by Okasaki, Lee, Tarditi [35] was latter shown to be non-normalizing on simply-typed terms [5].

In [5], the authors apply the methodology of Danvy's semantics artifacts to mechanically derive a continuation-passing style translation from a sequent calculus presentation of classical call-by-need. Starting from a specific evaluation strategy for the $\bar{\lambda}\mu\tilde{\mu}$-calculus, they finally obtain a small-step sequent calculus, the $\overline{\lambda}_{[lv\tau\star]}$-calculus, from which they get an untyped CPS translation almost for free[6]. The $\overline{\lambda}_{[lv\tau\star]}$-calculus can be understood as a refinement of the $\bar{\lambda}\mu\tilde{\mu}$-calculus with explicit environments (see Section 2.2). Before introducing our variant of this calculus using De Bruijn levels, let us first stick to names in order to emphasize the main differences between both calculi, which are:

- a new binder, written $\tilde{\mu}[x].\langle x \parallel F\rangle\tau'$, which is used to implement laziness as we shall explain;
- a subdivision of values (resp. co-values) into two categories of *weak* and *strong values* (resp. *catchable* and *forcing contexts*).

[6] See Appendix B.1 for the original definition of the $\overline{\lambda}_{[lv\tau\star]}$-calculus and the corresponding untyped CPS.

Strong values correspond to values strictly speaking. Weak values include variables, which force the evaluation of terms to which they refer into shared strong value. Their evaluation may require capturing a continuation. Dually, catchable contexts are co-values strictly speaking, while forcing contexts are contexts eagerly asking for a strong value, which may trigger the evaluation of terms lazily stored. In details, the lazy evaluation of terms allows for the following reduction:

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

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

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

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 environment $\tau$ which predates it, in front of the context $\tilde{\mu}[x].\langle x \parallel F \rangle \tau'$ where the $\tilde{\mu}[x]$ binder is waiting for a value. This context keeps trace of the part of the environment $\tau'$ 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 $\langle x \parallel F \rangle$ is evaluated in the updated full environment:

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

The brackets in $\tilde{\mu}[x].c$ are used to express the fact that the variable $x$ is forced at top-level[7]. 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. As such, reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus are close to the ones of the MAD[8].

Last but not least, the different syntactic categories can be understood as the different levels of alternation in a context-free abstract machine (see [5]): 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 untyped continuation-passing style defined in [5] (see Appendix B.1), and will thus be involved in the definition of our typed translation as well.

**De Bruijn levels** Ariola *et al.*'s presentation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus deeply relies on the assumption that names of variable are unique and thus on the possibility of performing $\alpha$-conversion on-the-fly[9]. In turns, we will use de Bruijn levels[10]

[7] Unlike meta-contexts of the shape $\tilde{\mu}x.C[\langle x \parallel F \rangle]$ in the $\overline{\lambda}_{lv}$-calculus, see [5].

[8] In fact, it is an easy exercise to simulate the MAD within the $\overline{\lambda}_{[lv\tau\star]}$-calculus. We outline this construction in Appendix A.

[9] See Appendix B.2 for more details on problems related to $\alpha$-conversion

[10] De Bruijn levels were originally introduced as a reversed notations for de Bruijn indices [8]. While they have been much less used, levels have the significant benefits that variables referring to the same binder have the same name.

| | | | |
|---|---|---|---|
| **Strong values** | $v ::= \boldsymbol{k} \mid \lambda x_i.t$ | **Stores** | $\tau ::= \varepsilon \mid \tau[x_i := t] \mid \tau[\alpha_i := E]$ |
| **Weak values** | $V ::= v \mid x_i$ | **Commands** | $c ::= \langle t \parallel e \rangle$ |
| **Terms** | $t, u ::= V \mid \mu\alpha_i.c$ | **Closures** | $l ::= c\tau$ |

| | |
|---|---|
| **Forcing contexts** | $F ::= \boldsymbol{\kappa} \mid t \cdot E$ |
| **Catchable contexts** | $E ::= F \mid \alpha_i \mid \tilde{\mu}[x_i].\langle x_i \parallel F \rangle\tau$ |
| **Evaluation contexts** | $e ::= E \mid \tilde{\mu}x_i.c$ |

| | | | | |
|---|---|---|---|---|
| (Let) | $\langle t \parallel \tilde{\mu}x_i.c \rangle\tau$ | $\to$ | $c[x_n/x_i]\tau[x_n := t]$ | with $\lvert\tau\rvert = n$ |
| (Catch) | $\langle \mu\alpha_i.c \parallel E \rangle\tau$ | $\to$ | $c[\alpha_n/\alpha_i]\tau[\alpha_n := E]$ | with $\lvert\tau\rvert = n$ |
| (Lookup$_\alpha$) | $\langle V \parallel \alpha_n \rangle\tau$ | $\to$ | $\langle V \parallel \tau(n) \rangle\tau$ | |
| (Lookup$_x$) | $\langle x_n \parallel F \rangle\tau[x_n := t]\tau'$ | $\to$ | $\langle t \parallel \tilde{\mu}[x_n].\langle x_n \parallel F \rangle\tau' \rangle\tau$ | |
| (Restore) | $\langle V \parallel \tilde{\mu}[x_i].\langle x_i \parallel F \rangle\tau' \rangle\tau$ | $\to$ | $\langle V \parallel \uparrow_n^{+i} F \rangle\tau[x_n := V](\uparrow_n^{+i} \tau')$ | with $\lvert\tau\rvert = n$ |
| (Beta) | $\langle \lambda x_i.t \parallel u \cdot E \rangle\tau$ | $\to$ | $\langle u \parallel \tilde{\mu}x_i.\langle t \parallel E \rangle \rangle\tau$ | |

**Fig. 4.** The $\overline{\lambda}_{[lv\tau\star]}$-calculus with de Bruijn levels

for variables (and co-variables) that are bound in the environment. Just as de Bruijn indices are pointers to the correct binder, de Bruijn levels are pointers to the correct cell of the environment. We use the mixed notation[11] $x_i$ where the relevant information is $x$ when the variable is bound within a proof (that is by a binder $\lambda$ or $\tilde{\mu}$), and where the relevant information is $i$ once the variable has been bound in the environment (at position $i$). For binders of evaluation contexts, we similarly use de Bruijn levels, but with variables of the form $\alpha_i$, where, again, $\alpha$ is a fixed name indicating that the variable is binding evaluation contexts, and the relevant information is the index $i$. The corresponding syntax is given in Figure 4, where the presence of names in the environments is absolutely useless and only there for readability.

As the environment can be dynamically extended during the execution, the location of a term in the environment and the corresponding pointer are likely to evolve (monotonically). Therefore, we need to be able to update de Bruijn levels within terms (contexts, etc.). To this end, we define the lifted term $\uparrow_n^{+i} t$ as the term $t$ where the free variables $x_j$ (resp. $\alpha_j$) with $j > n$ have been replaced by $x_{j+i}$ (resp. $\alpha_{i+j}$)[12]. The reduction rules are given in Figure 4. Note that we choose to perform indices substitutions as soon as they come, maintaining the property that $x_n$ always refers to the $(n+1)^{\text{th}}$ element of the environment.

Regarding the type system, we consider nine kinds of (monolatere) 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 $v$, $V$, $t$, $F$, $E$, $e$, $l$, $c$, $\tau$:

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

[11] Note that we could also use usual de Bruijn indices for bound variables within terms.
[12] See Appendix B.3 for a formal definition

$$\frac{(\boldsymbol{k} : T) \in \mathcal{S}}{\Gamma \vdash_v \boldsymbol{k} : T} \qquad \frac{\Gamma, x_n : T \vdash_t t : U \quad |\Gamma| = n}{\Gamma \vdash_v \lambda x_n.t : T \to U} \qquad \frac{\Gamma(n) = (x_n : T)}{\Gamma \vdash_V x_n : T} \qquad \frac{\Gamma \vdash_v v : T}{\Gamma \vdash_V v : T}$$

$$\frac{(\boldsymbol{\kappa} : T) \in \mathcal{S}}{\Gamma \vdash_F \boldsymbol{\kappa} : T^{\perp\!\!\!\perp}} \qquad \frac{\Gamma \vdash_t t : T \quad \Gamma \vdash_E E : U^{\perp\!\!\!\perp}}{\Gamma \vdash_F t \cdot E : (T \to U)^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, \alpha_n : T^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_t \mu\alpha_n.c : T} \qquad \frac{\Gamma \vdash_F F : T^{\perp\!\!\!\perp}}{\Gamma \vdash_E F : T^{\perp\!\!\!\perp}}$$

$$\frac{\Gamma(n) = (\alpha_n : T^{\perp\!\!\!\perp})}{\Gamma \vdash_E \alpha_n : T^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, x_i : T, \Gamma' \vdash_F F : T^{\perp\!\!\!\perp} \quad \Gamma, x_i : T \vdash_\tau \tau : \Gamma' \quad |\Gamma| = i}{\Gamma \vdash_E \tilde{\mu}[x_i].\langle x_i \parallel F\rangle\tau : T^{\perp\!\!\!\perp}} \qquad \frac{\Gamma \vdash_V V : T}{\Gamma \vdash_t V : T}$$

$$\frac{\Gamma, \alpha_n : T^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_t \mu\alpha_n.c : T} \qquad \frac{\Gamma \vdash_E E : T^{\perp\!\!\!\perp}}{\Gamma \vdash_e E : T^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, x_n : T \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_e \tilde{\mu}x_n.c : T^{\perp\!\!\!\perp}} \qquad \frac{}{\Gamma \vdash_\tau \varepsilon : \varepsilon}$$

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_t t : T \quad |\Gamma, \Gamma'| = n}{\Gamma \vdash_\tau \tau[x_n := t] : \Gamma', x_n : T} \qquad \frac{\Gamma \vdash_t t : T \quad \Gamma \vdash_e e : T^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \parallel e\rangle}$$

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_E E : T^{\perp\!\!\!\perp} \quad |\Gamma, \Gamma'| = n}{\Gamma \vdash_\tau \tau[\alpha_n := E] : \Gamma', \alpha_n : T^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, \Gamma' \vdash_c c \quad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}$$

**Fig. 5.** Typing rules for the $\overline{\lambda}_{[lv\tau\star]}$-calculus with de Bruijn levels

where types and typing contexts are defined by:

$$T, U ::= X \mid T \to U \qquad\qquad \Gamma ::= \varepsilon \mid \Gamma, x : T \mid \Gamma, \alpha : T^{\perp\!\!\!\perp}$$

Sequents typing values and terms are asserting a type, with the type written on the right; sequents typing contexts are expecting a type $T$ with the type written $T^{\perp\!\!\!\perp}$; sequents typing commands and closures are black boxes neither asserting nor expecting a type; sequents typing environments are instantiating a typing context.

The typing rules are given on Figure 5 where we adopt the convention that constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$ come with a signature $\mathcal{S}$ which assigns them a type. The typing rules are the same as for the named calculus [32], except for the one where indices should now match the length of the typing context. As in the named case, 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.* The proof proceeds by induction on typing derivation, and is almost the same as in the case without de Bruijn levels [32, Theorem 1].

## 3 Quantifying on environment extension: System $\mathbf{F}_{\boldsymbol{\Upsilon}}$

We shall now introduce System $F_\Upsilon$, the calculus that we will use as the target of several continuation-and-environment-passing style translations. We insist on

the fact that our purpose is to isolate the minimal ingredients that are necessary to define a generic target calculus, independently of the source languages (here simply typed calculi). We shall warn the reader that our interest in a minimal and generic system has the trade-off that its definition is quite technical. To ease the introduction of System $F_{\Upsilon}$, we first describe the intuitions guiding the translation of types (which we only outline in this section), highlighting the different features that it requires. As we will explain, the definition of $F_{\Upsilon}$ will be partially parameterized by the translation of formulas, making it usable for translating different calculi with global environments as we will see in Section 4.

### 3.1 Guidelines of the translation

Before presenting in detail the target system of the translation, let us explain step by step the rationale guiding the definition of the translation. Because of the sharing of the evaluation of arguments, the environment associating terms to variables has to be passed around. Passing the environment amounts to combining the continuation-passing style translation with an environment-passing style translation. Additionally, the environment is extensible, so, to anticipate extension of the environment, Kripke style forcing has to be used too, in a way comparable to what is done in step-indexing translations. To facilitate the comprehension of the different steps, we illustrate each of them with the translation of the sequent[13] $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$.

*Step 1 – Continuation-passing style.* In a first approximation, let us look only at the continuation-passing style part of the translation of a $\overline{\lambda}_{[lv\tau\star]}$ sequent.

As shown in [5] and as emphasized by the definition of the realizability interpretation in [32] reflecting the 6 nested syntactic categories used to define $\overline{\lambda}_{[lv\tau\star]}$, there are 6 different levels of control in call-by-need, leading to 6 mutually defined levels of interpretation. We define $[\![A \to B]\!]_v$ for strong values as $[\![A]\!]_t \to [\![B]\!]_E$, we define $[\![A]\!]_F$ for forcing contexts as $\neg\, [\![A]\!]_v$, $[\![A]\!]_V$ for weak values as $\neg\, [\![A]\!]_F =^{2}_{\neg} [\![A]\!]_v$, and so on until $[\![A]\!]_e$ defined as $\stackrel{5}{\neg}\, [\![A]\!]_v$ (where we use the notations $\stackrel{1}{\neg} A \triangleq A \to \bot$ and $^{\mathrm{n}\pm 1}_{\neg}\, A \triangleq \neg\, \stackrel{\mathrm{n}}{\neg} A$).

As observed in the realizability interpretation [32], hypotheses from a context $\Gamma$ of the form $\alpha : A^{\perp\!\!\!\perp}$ are to be translated as $[\![A]\!]_E =^{3}_{\neg} [\![A]\!]_v$ while hypotheses of the form $x : A$ are to be translated as $[\![A]\!]_t =^{4}_{\neg} [\![A]\!]_v$. Up to this point, if we denote this translation of $\Gamma$ by $[\![\Gamma]\!]$, in the particular case of $\Gamma \vdash_t A$ the translation is $[\![\Gamma]\!] \vdash [\![A]\!]_t$ and similarly for other levels.

*Example 1 (Translation, step 1).* Up to now, the translation taking into account the continuation-passing style of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ is simply:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C]\!] &= a : [\![A]\!]_t \quad , \alpha : [\![A]\!]_E \quad , b : [\![B]\!]_t \quad \vdash [\![e]\!]_e : [\![C]\!]_e \\ &= a :^{4}_{\neg} [\![A]\!]_v \ , \alpha :^{3}_{\neg} [\![A]\!]_v \ , b :^{4}_{\neg} [\![B]\!]_v \vdash [\![e]\!]_e :^{5}_{\neg} [\![C]\!]_v\end{aligned}$$

[13] We omit de Bruijn levels for the moment and we write $a : A, b : B, ...$ instead of $x : T, y : U, ...$ for the sole purpose of easing readability.

*Step 2 – environment-passing style.* The continuation-passing style part being settled, the environment-passing style part should be considered. In particular, the translation of $\Gamma \vdash_t A$ is not anymore a sequent $[\![\Gamma]\!] \vdash [\![A]\!]_t$ but instead a sequent roughly of the form $\vdash [\![\Gamma]\!] \to [\![A]\!]_t$, with actually $[\![\Gamma]\!]$ being passed around not only at the top-level of $[\![A]\!]_t$ but also every time a negation is used. We write this sequent $\vdash [\![\Gamma]\!] \triangleright_t A$ where $\triangleright_t A$ is defined by induction on $t$ and $A$:

$$[\![\Gamma]\!] \triangleright_t A \;=\; [\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_E A) \to \bot \;=\; [\![\Gamma]\!] \to ([\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_V A) \to \bot) \to \bot \;\cdots$$

Moreover, the translation of each type in $\Gamma$ should itself be abstracted over the environment at each use of a negation.

*Example 2 (Translation, step 2).* Up to now, the continuation-and-environment-passing style translation of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ is:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C]\!] = \quad & \vdash [\![e]\!]_e : [\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] \triangleright_e C \\ = \quad & \vdash [\![e]\!]_e : [\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] \to ([\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] \triangleright_t C) \to \bot = ...\end{aligned}$$

where:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] &= [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \triangleright_t B \\ &= [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \to ([\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \triangleright_E B) \to \bot = ... \\ [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] &= [\![a : A]\!],\ \alpha : [\![a : A]\!] \triangleright_E A \\ &= [\![a : A]\!],\ \alpha : [\![a : A]\!] \to ([\![a : A]\!] \to \triangleright_E A) \to \bot = ... \\ [\![a : A]\!] &= a : \varepsilon \triangleright_t A \quad = \quad a :^{\neg\!\!\!\perp} [\![A]\!]_v\end{aligned}$$

*Step 3 – Extension of the environment* The environment-passing style part being settled, it remains to anticipate that the environment is extensible. This is done by supporting arbitrary insertions of any term at any place in the environment. The extensibility is obtained by quantifying over all possible extensions of the environment at each level of the negation. In the realizability model, this was reflected by the compatibility of realizers with any environment extension [32].

For this purpose, we use as a type system an adaptation of System $F_{<:}$ [9] extended with stores, defined as lists of assignations $[x := t]$. *Store types*, denoted by $\Upsilon$, are defined as list of types of the form $(x : A)$ where $x$ is a name and $A$ is a type properly speaking and admit a subtyping notion $\Upsilon' <: \Upsilon$ to express that $\Upsilon'$ is an extension of $\Upsilon$. This corresponds to the following refinement of the definition of $[\![\Gamma]\!] \triangleright_t A$:

$$\begin{aligned}[\![\Gamma]\!] \triangleright_t A &= \forall \Upsilon <: [\![\Gamma]\!].\Upsilon \to (\Upsilon \triangleright_E A) \to \bot \\ &= \forall \Upsilon <: [\![\Gamma]\!].\Upsilon \to (\forall \Upsilon' <: \Upsilon.\Upsilon' \to \Upsilon' \triangleright_V A \to \bot) \to \bot = ...\end{aligned}$$

Such a quantification is reminiscent of Kripke forcing [21]: thinking of store types $\Upsilon$ as *worlds*, the *accessible worlds* from $\Upsilon$ are precisely all the possible $\Upsilon' <: \Upsilon$. To emphasize this correspondence, we give here the translation of the arrow both in Kripke models and in our setting, where the forcing translation is interleaved with the continuation/environment-passing parts:

$$\begin{aligned}\Upsilon \triangleright_v T \to U \quad &\triangleq \quad \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t T) \to (Y \triangleright_E U) \to \bot \\ \omega \Vdash A \Rightarrow B \quad &\triangleq \quad \forall \omega' \geq \omega. \qquad \omega' \Vdash A \quad \Rightarrow \quad \omega' \Vdash B\end{aligned}$$

*Example 3 (Translation, step 3).* The translation, now taking into account store extensions, of $a : A, \alpha : A^{\perp\!\!\perp}, b : B \vdash_e e : C$ becomes:

$$\begin{aligned}\llbracket a : A, \alpha : A^{\perp\!\!\perp}, b : B \vdash_e e : C \rrbracket &= \ \vdash \llbracket e \rrbracket_e : \llbracket a : A, \alpha : A^{\perp\!\!\perp}, b : B \rrbracket \triangleright_e C \\ &= \ \vdash \llbracket e \rrbracket_e : \forall \Upsilon <: \llbracket a : A, \alpha : A^{\perp\!\!\perp}, b : B \rrbracket . \Upsilon \to (\Upsilon \triangleright_t C) \to \bot = ...\end{aligned}$$

where:

$$\begin{aligned}\llbracket a : A, \alpha : A^{\perp\!\!\perp}, b : B \rrbracket &= \llbracket a : A, \alpha : A^{\perp\!\!\perp} \rrbracket,\ b : \llbracket a : A, \alpha : A^{\perp\!\!\perp} \rrbracket \triangleright_t B \\ &= \llbracket a : A, \alpha : A^{\perp\!\!\perp} \rrbracket,\ b : \forall \Upsilon <: \llbracket a : A, \alpha : A^{\perp\!\!\perp} \rrbracket . \Upsilon \to (\Upsilon \triangleright_E B) \to \bot = ... \\ \llbracket a : A, \alpha : A^{\perp\!\!\perp} \rrbracket &= \llbracket a : A \rrbracket,\ \alpha : \llbracket a : A \rrbracket \triangleright_E A \\ &= \llbracket a : A \rrbracket,\ \alpha : \forall \Upsilon <: \llbracket a : A \rrbracket . \Upsilon \to (\Upsilon \triangleright_V A) \to \bot = ... \\ \llbracket a : A \rrbracket &= a : \varepsilon \triangleright_t A \quad = \quad a : \forall \Upsilon . \Upsilon \to (\Upsilon \triangleright_E A) \to \bot\end{aligned}$$

The only remaining step is to take into account de Bruijn levels both inside the source and target languages. Therefore, in the target language stores are simply defined as lists of terms while store types are lists of types (that is to say $A, A^{\perp\!\!\perp}, \dots$ instead of $a : A, \alpha : A^{\perp\!\!\perp}, \dots$). Interestingly, de Bruijn levels will give a computational content to the subtyping relation $\Upsilon' <: \Upsilon$ through explicit coercions $\sigma : \Upsilon' <: \Upsilon$ mapping each type in $\Upsilon$ to the corresponding type in $\Upsilon'$.

### 3.2 System $F_\Upsilon$

The target language of the continuation-and-environment-passing style translations we will present in the next section thus includes the different features that we previously evoked. Namely, the target language is defined as the usual $\lambda$-calculus extended with stores (that are lists of terms) and coercions to keep track of store extension and to update pointers. As for its type system, it contains simple types as in the source language and a bounded second-order quantification over store types. We refer to this language as System $F_\Upsilon$ which syntax of terms is given by:

$$\begin{array}{lll} & t, u & ::= \boldsymbol{k} \mid x \mid \lambda x.t \mid tu \mid \lambda s.t \mid t\sigma \mid \lambda \delta.t \mid t\tau \\ \textbf{(Terms)} & & \mid \ \texttt{split}\ \tau\ \texttt{at}\ n\ \texttt{along}\ \sigma\ \texttt{as}\ \delta_1, x, \delta_2\ \texttt{in}\ t \\ \textbf{(Coercions)} & \sigma & ::= \varepsilon \mid s \mid \sigma^+ \mid \Uparrow \sigma \mid \sigma' \circ \sigma \\ \textbf{(Stores)} & \tau & ::= [] \mid \delta \mid [t]_t \mid [t]_E \mid \tau\tau' \end{array}$$

while types are defined by the following grammar:

$$\begin{array}{lll} \textbf{(Types)} & A, B & ::= X \mid A \to B \mid \forall Y <: \Upsilon . A \mid \Upsilon \triangleright_\tau \Upsilon' \to A \\ \textbf{(Store types)} & \Upsilon & ::= \emptyset \mid \Upsilon, Y \mid \Upsilon, F \\ \textbf{(Simple types)} & T, U & ::= X \mid T \to U \qquad F ::= T, T^{\perp\!\!\perp} \\ \textbf{(Typing contexts)} & \Gamma & ::= \cdot \mid \Gamma, x : A \mid \Gamma, s : Y <: \Upsilon \mid \Gamma, \delta : \Upsilon \triangleright_\tau \Upsilon' \end{array}$$

As we shall explain thereafter, the variable $Y$ is actually a shorthand for a vector of second-order variables. We also assume that types contain at least a constant for each atomic type $X$ of the original system, and we still denote this constant

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A}\,(\text{Ax}) \qquad \frac{(\boldsymbol{k}:A)\in\mathcal{S}}{\Gamma\vdash \boldsymbol{k}:A}\,(c) \qquad \frac{\Gamma,x:A\vdash t:B}{\Gamma\vdash\lambda x.t:A\to B}\,(\lambda) \qquad \frac{\Gamma\vdash t:A\to B \quad \Gamma\vdash u:A}{\Gamma\vdash t\,u:B}\,(@)$$

$$\frac{\Gamma,\delta:\Upsilon_0\rhd_\tau\Upsilon_1\vdash t:B}{\Gamma\vdash\lambda\delta.t:\Upsilon_0\rhd_\tau\Upsilon_1\to B}\,(\tau_I) \qquad \frac{\Gamma\vdash t:\Upsilon_0\rhd_\tau\Upsilon_1\to B \quad \Gamma\vdash\tau:\Upsilon_0\rhd_\tau\Upsilon_1}{\Gamma\vdash t\,\tau:B}\,(\tau_E)$$

$$\frac{\Gamma,s:Y<:\Upsilon\vdash t:A \quad Y\notin FV(\Gamma)}{\Gamma\vdash\lambda s.t:\forall Y<:\Upsilon.A}\,(\forall_I) \qquad \frac{\Gamma\vdash t:\forall Y<:\Upsilon.A \quad \Gamma\vdash\sigma:\Upsilon'<:\Upsilon}{\Gamma\vdash t\,\sigma:A\{Y:=\Upsilon'\}}\,(\forall_E)$$

$$\frac{\begin{array}{l}\Gamma\vdash\tau:\Upsilon_0,T,\Upsilon_1\\ \Gamma\vdash\sigma:\Upsilon_0',T,\Upsilon_1'<:\Upsilon_0,T,\Upsilon_1\end{array} \qquad \Gamma,\delta_0:\Upsilon_0',x:\Upsilon_0'\rhd_t T,\delta_1:\Upsilon_0',T\rhd_\tau\Upsilon_1'\vdash t:B}{\Gamma\vdash\texttt{split}\ \tau\ \texttt{at}\ |\Upsilon_0|\ \texttt{along}\ \sigma\ \texttt{as}\ \delta_0,x,\delta_1\ \texttt{in}\ t:B}\,(\texttt{split})$$

$$\frac{\begin{array}{l}\Gamma\vdash\tau:\Upsilon_0',T^{\perp\!\!\!\perp},\Upsilon_1'\\ \Gamma\vdash\sigma:\Upsilon_0',T^{\perp\!\!\!\perp},\Upsilon_1'<:\Upsilon_0,T^{\perp\!\!\!\perp},\Upsilon_1\end{array} \qquad \Gamma,\delta_0:\Upsilon_0',x:\Upsilon_0'\rhd_E T,\delta_1:\Upsilon_0',T^{\perp\!\!\!\perp}\rhd_\tau\Upsilon_1'\vdash t:B}{\Gamma\vdash\texttt{split}\ \tau\ \texttt{at}\ |\Upsilon_0|\ \texttt{along}\ \sigma\ \texttt{as}\ \delta_0,x,\delta_1\ \texttt{in}\ t:B}\,(\texttt{split}^{\perp\!\!\!\perp})$$

---

$$\frac{}{\Gamma\vdash[]:\emptyset\rhd_\tau\emptyset}\,(\varepsilon) \qquad \frac{\Gamma\vdash t:\Upsilon\rhd_t T}{\Gamma\vdash[t]_t:\Upsilon\rhd_\tau T}\,(\tau_t) \qquad \frac{\Gamma\vdash t:\Upsilon\rhd_E T}{\Gamma\vdash[t]_E:\Upsilon\rhd_\tau T^{\perp\!\!\!\perp}}\,(\tau_E)$$

$$\frac{(\delta:\Upsilon_0\rhd_\tau\Upsilon)\in\Gamma}{\Gamma\vdash\delta:\Upsilon_0\rhd_\tau\Upsilon}\,(\tau_{\texttt{ax}}) \qquad \frac{\Gamma\vdash\tau:\Upsilon_0\rhd_\tau\Upsilon \qquad \Gamma\vdash\tau':(\Upsilon_0,\Upsilon)\rhd_\tau\Upsilon'}{\Gamma\vdash\tau\tau':\Upsilon_0\rhd_\tau\Upsilon,\Upsilon'}\,(\tau\tau')$$

---

$$\frac{}{\Gamma\vdash\varepsilon:\emptyset<:\emptyset}\,(<:_\varepsilon) \qquad \frac{(s:\Upsilon'<:\Upsilon)\in\Gamma}{\Gamma\vdash s:\Upsilon'<:\Upsilon}\,(<:_{\texttt{ax}}) \qquad \frac{\Gamma\vdash\sigma:\Upsilon'<:\Upsilon}{\Gamma\vdash\sigma^+:(\Upsilon',F)<:(\Upsilon,F)}\,(<:_+)$$

$$\frac{\Gamma\vdash\sigma:\Upsilon'<:\Upsilon}{\Gamma\vdash\Uparrow\sigma:(\Upsilon',F)<:\Upsilon}\,(<:_\Uparrow) \qquad \frac{\Gamma\vdash\sigma:\Upsilon'<:\Upsilon \quad \Gamma\vdash\sigma':\Upsilon''<:\Upsilon'}{\Gamma\vdash\sigma'\circ\sigma:\Upsilon''<:\Upsilon}\,(<:_\circ)$$

**Fig. 6.** Typing rules of System $F_\Upsilon$

by $X$. This allows us to define an embedding $\iota$ from the original type system to this one by:

$$\iota(X)=X \qquad\qquad \iota(T\to U)=\iota(T)\to\iota(U)$$

To alleviate notations, we will identify $\iota(T)$ and $T$ in the sequel of the paper.

Regarding the reduction rules of the language, there are only four of them, three witnessing the usual $\beta$-reduction (for application to terms, stores and coercions) and one to split stores with respect to a given index:

$$(\lambda x.t)\,u\to t[u/x] \qquad (\lambda\delta.t)\,\tau\to t[\tau/\delta] \qquad (\lambda s.t)\,\sigma\to t[\sigma/s]$$
$$\texttt{split}\ \tau_0[u]\tau_1\ \texttt{at}\ n\ \texttt{along}\ \sigma\ \texttt{as}\ \delta_1,x,\delta_2\ \texttt{in}\ t\to t[\tau_0/\delta_0,u/x,\tau_1/\delta_1]$$

where $|\tau_0|=[\![\sigma]\!](n)$. For the last rule, we assume that the coercion $\sigma$ is computable (that is without variables) and $[\![\sigma]\!]$ refers to the partial function from $\mathbb{N}$ to $\mathbb{N}$ associated to $\sigma$. We will come back to this in the next section, we refer the reader to Example 4 for a comprehensive example on coercions.

We shall attract the reader's attention on several aspects of the typing rules, which are given in Figure 6 and include three kinds of typing sequents: one for terms, one for stores and one for coercions. First, observe that we introduce a new symbol $\Upsilon \triangleright_\tau \Upsilon'$ to denote the fact that a store has a type conditioned by $\Upsilon$ (which should be the type of the (missing) beginning of the store). In order to ease the notations, we will sometimes write $\Upsilon$ instead of $\emptyset \triangleright_\tau \Upsilon$ in the sequel.

Second, notice that the typing rules to form and split stores are parameterized by two translations of simple types written $\Upsilon \triangleright_t T$ and $\Upsilon \triangleright_E T$. These notations are shorthands which we will define later when typing the continuation-and-environment-passing style translation in Section 4 (we will for instance translate a sequent $\Gamma \vdash_t t : T$ by $\vdash [\![t]\!]_t : [\![\Gamma]\!] \triangleright_t T$). In particular, this will allow us to use the same type system for different translations, since $\Upsilon \triangleright_t T$ will be defined differently depending on whether we consider a call-by-need or call-by-name translation. Incidentally, this also means that the rules to type stores force elements of the store to have types following the structure of types obtained through the translations. Even though this could appear as a strong requirement, it appears naturally when using coercion to witness the inclusion $\Upsilon' <: \Upsilon$. Indeed, with de Bruijn levels we need to update pointers when inserting a new element, and such an operation would not have any sense (and in particular would be ill-typed) for an element that is not of type $\Upsilon \triangleright_t T$[14]. Besides, note that each element of the store has a type depending on the type of the head of the store. Once again, this is natural and only reflects what was already happening in the source language or within the realizability interpretation.

Last, but not least, we shall explain that the quantification $\forall Y <: \Upsilon.A$ is actually a (voluntarily) simplifying notations hiding the need for considering vectors of second order variables. Indeed, remember that we need to anticipate arbitrary insertions of any terms at any place. For instance, this means that a store $[t]_t[u]_E$ of type $\triangleright_\tau T, U^{\perp\!\!\!\perp}$ might be arbitrarily extended into a store $\tau_0[t]_t\tau_1[u]_E\tau_2$ of type $\triangleright_\tau \boldsymbol{T_0}, T, \boldsymbol{T_1}, U^{\perp\!\!\!\perp}, \boldsymbol{T_2}$ where $\boldsymbol{T}_i = T_{i0}, \dots, T_{in}$ and where we write $\boldsymbol{T_0}, T, \boldsymbol{T_1}$ for the flatten list $T_{00}, T_{01}, \dots, T, T_{10}, \dots$. Therefore, we should strictly speaking consider the second-order variable $Y$ in the quantification $\forall Y <: \Upsilon.A$ as a vector of vector of types, whose length is fully determine by the one of $\Upsilon$. Writing $\boldsymbol{Y}^p_k$ for the vector of types $Y_{k0}, \dots, Y_{kp}$, we should then formally define $\forall Y <: \Upsilon.A$ as:

$$\forall \boldsymbol{Y}^{p_0} \dots \forall \boldsymbol{Y}^{p_n}.(Y_0^{p_0}\Upsilon(0)Y_1^{p_1}\Upsilon(1)\dots Y_n^{p_n}) <: \Upsilon \to A$$

where $n = |\Upsilon|$. Observe in particular that the vector of vectors $\boldsymbol{Y}^{|\Upsilon|} = \boldsymbol{Y}^{p_0}, \dots, \boldsymbol{Y}^{p_n}$ is interleaved with the types in $\Upsilon$, so that we only quantify over the extensions while the type of the extended store is defined by interleaving the quantified types and the existing ones.

We do not want to enter here into too many details about vectors and the definition of our subtyping relation, but readers familiar with type theory should easily be convinced that both are perfectly definable in any language with a form

[14] One could circumvent this by tagging each cell of the store with a flag (using a sum type) indicating whether the corresponding elements have a type of this form or not, but here it would only make the translation more complex.

of dependent types allowing for the definition of (sized) vectors. In particular, such a structure could be defined in Coq or Agda without any further difficulties. Even though considering the size of vectors will be crucial to ensure the correctness of the definition in Figure 6, to ease the notations we will omit them most of the time and stick to our simplified quantification $\forall Y <: \Upsilon$. For most of the definitions and proofs, it is enough to know that any store type has a size (which might depend on some integers), and that consequently, any typed store $\tau : \Upsilon$ also has a size which coincides with the one of its type[15].

### 3.3 Properties

We shall now state a few properties of system $F_\Upsilon$, and in particular of coercions.

**Properties of system $F_\Upsilon$** We begin with stating basic properties of the system that are independent of the translations $\Upsilon \rhd_t T$ and $\Upsilon \rhd_E T$ that parameterize the type system. First, it is clear that the type system is compatible with the following weakening rule:

$$\frac{\Gamma \vdash t : A \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash t : A} \ (\Gamma_w)$$

We can verify the safety of types with respect to reductions:

**Proposition 2 (Subject reduction)** *For any context $\Gamma$, any type $A$ and any terms $t, t'$, if $\Gamma \vdash t : A$ and $t \to t'$, then $\Gamma \vdash t' : A$.*

*Proof.* See Appendix D.

**Normalization** It should be clear to the reader that system $F_\Upsilon$ could be defined through a more expressive language including a form a dependent types and a second-order quantification. In particular, it could be expressed within many type theories or proof assistants, which especially implies that typed terms are normalizing. Combined this with Ariola *et al.*'s result of operational correctness for the CPS translation[16], this would provide us with an normalization proof for the $\overline{\lambda}_{[lv\tau\star]}$-calculus. Once again, we could make this statement more formal but as we already dispose of a proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we prefer to focus on the details of the different translations.

**Coercions** We say that a coercion is *in normal form* if it contains neither variables nor compositions of coercions. Formally, these coercions are given by the following grammar:

*Normal forms* $\qquad \sigma ::= \varepsilon \mid \sigma^+ \mid \ \Uparrow\sigma$

[15] In particular, when considering a variable $\delta : Y$, we define $|\delta|$ as $|Y|$.

[16] Indeed, once the coercions erased, our translation is essentially Ariola *et al.*'s translation, hence we could benefit from their result of computational correctness.

Interestingly, when two coercions are in normal form, it is possible to compose them to compute another normal form. In other word, we can define the composition function $- \odot -$ between coercions in normal form by:

$$\begin{array}{ll} \sigma_1^+ \odot \sigma_0^+ \triangleq (\sigma_1 \odot \sigma_0)^+ & \Uparrow\sigma_1 \odot \sigma_0 \triangleq \Uparrow(\sigma_1 \odot \sigma_0) \\ \sigma_1^+ \odot \Uparrow\sigma_0 \triangleq \Uparrow(\sigma_1 \odot \sigma_0) & \varepsilon \odot \sigma_0 = \sigma_1^+ \odot \varepsilon \triangleq \varepsilon \end{array}$$

We can verify that this function is sound with respect to the typing rule for composing coercions:

**Lemma 3 (Composition of normal forms)** *If $\sigma, \sigma'$ are coercions in normal forms such that $\vdash \sigma : \Upsilon <: \Upsilon'$ and $\vdash \sigma' : \Upsilon' <: \Upsilon''$, then $\vdash \sigma' \odot \sigma : \Upsilon <: \Upsilon''$.*

*Proof.* Direct by structural induction on $\sigma'$.

This lemma suggests us that we can actually consider a slightly larger fragment that includes compositions, since we are able to compute them to get normal form. We define *computable coercions* as being coercions of the shape:

**Computable coercions** $\sigma ::= \varepsilon \mid \sigma^+ \mid \ \Uparrow\sigma \mid \sigma' \circ \sigma$

We can in fact safely reduce properties of computable coercions to the ones of normal forms:

**Proposition 4 (Computing normal forms)** *For any computable $\sigma$, if $\Gamma \vdash \sigma : \Upsilon' <: \Upsilon$ then there exists $\sigma_n$ in normal form such that $\Gamma \vdash \sigma_n : \Upsilon' <: \Upsilon$.*

*Proof.* By induction on typing derivations, using the previous lemma for $\sigma' \circ \sigma$.

It is worth noting that for any $\sigma, \Upsilon, \Upsilon'$, if $\vdash \sigma : \Upsilon' <: \Upsilon$, then $\sigma$ is necessarily computable. We then deduce as a corollary of the previous proposition:

**Corollary 5** *If $\vdash \sigma : \Upsilon' <: \Upsilon$, then $|\Upsilon| \leq |\Upsilon'|$.*

*Proof.* Using the previous proposition, we can reduce this to the case of $\sigma$ in normal forms. The proof then proceed by easy structural induction on $\sigma$.

If a computable coercion is typed by $\vdash \sigma : \Upsilon' <: \Upsilon$, we can actually identify it with a partial monotone function from $[0, |\Upsilon|]$ to $[0, [\Upsilon'|]$ which intuitively maps the index of every type in $\Upsilon$ to its corresponding index in $\Upsilon'$.

Formally, if $\sigma$ is a coercion in normal form (if it is computable we first reduce it to a computation in normal form), we define its domain $\mathrm{dom}(\sigma)$ and its codomain $\mathtt{codom}(\sigma)$ by:

$$\begin{array}{rcl|rcl|rcl} \mathrm{dom}(\varepsilon) & \triangleq & 0 & \mathrm{dom}(\sigma^+) & \triangleq & \mathrm{dom}(\sigma) + 1 & \mathrm{dom}(\Uparrow\sigma) & \triangleq & \mathrm{dom}(\sigma) \\ \mathtt{codom}(\varepsilon) & \triangleq & 0 & \mathtt{codom}(\sigma^+) & \triangleq & \mathtt{codom}(\sigma) + 1 & \mathtt{codom}(\Uparrow\sigma) & \triangleq & \mathtt{codom}(\sigma) + 1 \end{array}$$

We then associate to $\sigma$ the partial function $[\![\sigma]\!]$ from $[0, \mathrm{dom}(\sigma)]$ to $[0, \mathtt{codom}(\sigma)]$ defined by:

$$\begin{array}{l} [\![\varepsilon]\!] \triangleq \{0 \mapsto 0\} \\ [\![\sigma^+]\!] \triangleq [\![\sigma]\!] \cup \{\mathrm{dom}(\sigma) \mapsto \mathtt{codom}(\sigma)\} \end{array} \quad \left| \quad [\![\Uparrow\sigma]\!] \triangleq \begin{cases} n < \mathrm{dom}(\sigma) \mapsto [\![\sigma]\!](n) \\ n = \mathrm{dom}(\sigma) \mapsto [\![\sigma]\!](n) + 1 \end{cases} \right.$$

Notice that $[\![\sigma]\!]$ is always a strictly monotone function. We can check that these definitions are indeed in adequacy with the intuition we gave above:

**Proposition 6 (Associated function)** *If $\sigma$ is in normal form and s.t. $\vdash \sigma : \Upsilon' <: \Upsilon$, then:*

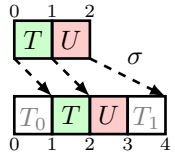
1. $\mathrm{dom}(\sigma) = |\Upsilon|$
2. $\mathtt{codom}(\sigma) = |\Upsilon'|$
3. $\forall n < |\Upsilon|, \Upsilon'([\![\sigma]\!](n)) = \Upsilon(n)$
4. $[\![\sigma]\!](|\Upsilon|) = |\Upsilon'|$

*Proof.* See Appendix D.

This proposition thus opens the way for proving properties of computable coercions through their associated functions.

*Example 4.* As an example, we let the reader verify that for any types $T, U, T_0, T_1$, if we denote by $\sigma$ for the coercion $\Uparrow(\Uparrow\varepsilon)^{++}$ which is in normal form, we have:

- $\vdash \Uparrow(\Uparrow\varepsilon)^{++}) : T_0, T, U, T_1 <: T, U$
- $\mathrm{dom}(\sigma) = 2$, $\mathtt{codom}(\sigma) = 4$
- $[\![\sigma]\!] : \begin{cases} 0 \mapsto 1 \\ 1 \mapsto 2 \\ 2 \mapsto 4 \end{cases}$




Visually, this corresponds to the situation pictured on the right.

Last, we introduce the following shorthands:

$$\sigma^{0+} \triangleq \sigma \qquad \sigma^{(k+1)+} \triangleq (\sigma^+)^{k+} \qquad \Uparrow^0\sigma \triangleq \sigma \qquad \Uparrow^{k+1}\sigma \triangleq \Uparrow^k(\Uparrow\sigma)$$
$$\mathtt{id}_0 = \emptyset \qquad \mathtt{id}_{n+1} = (\mathtt{id}_n)^+ \qquad \mathtt{shift}_n^k \triangleq \Uparrow^k \mathtt{id}_n$$

that will be useful when defining the translations. Indeed, we have that:

**Lemma 7 (Shifts)** *For any $\Upsilon_0, \Upsilon_0', \Upsilon_1$, writing $n = |\Upsilon_0|$ and $k = |\Upsilon_1|$ we have:*

$$\frac{\Gamma \vdash \sigma : \Upsilon_0' <: \Upsilon_0}{\Gamma \vdash \sigma^{k+} : \Upsilon_0'\Upsilon_1 <: \Upsilon_0\Upsilon_1} \qquad \frac{}{\Gamma \vdash \mathtt{shift}_n^k : \Upsilon_0\Upsilon_1 <: \Upsilon_0}$$

*Proof.* Both proofs are easy inductions on $|\Upsilon_1|$.

**Lifting terms and stores** Finally, we conclude this section by showing how terms and stores can be lifted using coercions for their types to remain consistent while extended stores are passed in the translations. First, we show that the bounded quantification can be composed with a subtyping relation witnessed by a coercion $\sigma$, by precomposing terms with $\sigma$. Given a coercion $\sigma$ of type $\Upsilon' <: \Upsilon$ and a term $t$ whose type is of the shape $\forall Y <: \Upsilon.A$, we define:

$$(\uparrow^\sigma t) \triangleq \lambda s.t\ (s \circ \sigma)$$

**Lemma 8** *The following rule is admissible:*

$$\frac{\Gamma \vdash t : \forall Y <: \Upsilon.A \quad \Gamma \vdash \sigma : \Upsilon' <: \Upsilon}{\Gamma \vdash (\uparrow^\sigma t) : \forall Y <: \Upsilon'.A}\ (\uparrow^\sigma)$$

*Proof.* See Appendix D.

When the translations $\Upsilon \rhd_t T$ and $\Upsilon \rhd_E T$ are of the shape $\forall Y <: \Upsilon....$, this definition can be scaled to stores by setting:

$$\uparrow^\sigma(\varepsilon) \triangleq \varepsilon \qquad \uparrow^\sigma([t]_t\tau) \triangleq [\uparrow^\sigma t]_t(\uparrow^{\sigma^+}\tau) \qquad \uparrow^\sigma([t]_E\tau) \triangleq [\uparrow^\sigma t]_E(\uparrow^{\sigma^+}\tau)$$

Observe that in a store $[t]_t[u]_t\tau$, $t$ is lifted with $\sigma$ while $u$ is lifted with $\sigma^+$ (and so on recursively). This is due to the fact that if $\sigma$ is of type $\Upsilon' <: \Upsilon$ and the store of type $\Upsilon \rhd_\tau T, U, ...$, the term $t$ is then of type $\Upsilon \rhd_t T$ and can be lifted with $\sigma$. In turns, $u$ is of some type $\Upsilon, T \rhd_t U$ and thus requires to be lifted with a coercion of type $\Upsilon', T <: \Upsilon, T$, that is to say $\sigma^+$. More generally, we deduce from the former lemma the following corollary that will be crucial when typing the translation of terms:

**Corollary 9** *If $\Upsilon \rhd_t T$ (resp. $\Upsilon \rhd_E T$) is of the shape $\forall X <: \Upsilon.F(X, A)$ (resp. $\forall X <: \Upsilon.G(X, T)$), then the following rules is admissible:*

$$\frac{\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^\sigma \tau) : \Upsilon_1 \rhd_\tau \Upsilon}$$

*Proof.* See Appendix D.

## 4 Continuation-and-environment-passing style translations

We are now equipped to define typed continuation-and-environment-passing style translations to system $F_\Upsilon$. These translations exactly follows the intuitions outlined in Section 3.1. We first focus on the case of the (call-by-need) $\overline{\lambda}_{[lv\tau\star]}$-calculus, and then illustrate the generality of our method by giving a translation for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments. Theses translations also apply to the MAD and the MAM through the adequate embeddings (see Appendix A). A call-by-value translation is also given in Appendix G.

### 4.1 A typed call-by-need translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

**Translation of terms** We can now take advantage of the features of system $F_\Upsilon$ to type Ariola *et al.*'s untyped translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus (given in Figure 13). This translation was obtained by refining the reduction system (see Figure 11) into a context-free abstract machine (that is to say an abstract machine in which it can be decided which reduction rule to apply by analyzing only the term or the context independently) [5]. The translation of terms is nothing more than the untyped translation of Ariola *et al.* rephrased to handle De Bruijn levels and coercions. Along the translation, we maintain two invariants on de Bruijn levels:

1. Stores are always consistent, that is, in a store $\tau[t]_o\tau'$, $t$ has its levels coherent with its prefix $\tau$, and ignores its suffix $\tau'$. In terms of types, if $\tau$ is of type $\Upsilon$, $t$ will be of a type $\Upsilon \rhd_o -$.

$$[\![\lambda x_i.t]\!]_v\ \sigma\ \tau\ u\ E \triangleq [\![t]\!]_t\ \sigma^+\ \tau[u]\ \uparrow^{\mathtt{shift}^1_{|\tau|}} E \qquad [\![\boldsymbol{k}]\!]_v \triangleq \boldsymbol{k}$$

$$[\![t \cdot E]\!]_F\ \sigma\ \tau\ v \triangleq v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![t]\!]_t)\ (\uparrow^\sigma [\![E]\!]_E) \qquad [\![\boldsymbol{\kappa}]\!]_F \triangleq \boldsymbol{\kappa}$$

$$[\![v]\!]_V\ \sigma\ \tau\ F \triangleq F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![v]\!]_v)$$
$$[\![x_i]\!]_V\ \sigma\ \tau\ F \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1$$
$$\mathtt{in}\ x\ \mathtt{id}_{|\delta_0|}\ \delta_0\ (\lambda s \delta_0' V.V\ (\mathtt{shift}^p_{n+k})\ (\delta_0'[\uparrow^t V]_t \uparrow^{s^+} \delta_1)\ (\uparrow^{\sigma'} F))$$
$$\text{where } n = |\delta_0| = [\![\sigma]\!](i),\ k = |\delta_0'| - n,\ p = |\delta_1| + 1,\ \sigma' = s^{p+}$$
$$\text{and } \uparrow^t V = \lambda s \delta E.E\ \mathtt{id}_{|\delta|}\ \delta\ (\uparrow^s V)$$

$$[\![\alpha_i]\!]_E\ \sigma\ \tau\ V \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\ x\ (\mathtt{shift}^{|\delta_1|+1}_{|\delta_0|})\ \tau\ V$$
$$[\![\tilde{\mu}[x_i].\langle x_i \parallel F\rangle \tau']\!]_E\ \sigma\ \tau\ V \triangleq V\ \mathtt{shift}^{|\tau'|+1}_{|\tau|}\ \tau[\uparrow^t V]_t(\uparrow^{\sigma^+} [\![\tau']\!]_\tau)\ (\uparrow^{\sigma'} [\![F]\!]_F)$$
$$\text{where } k = |\tau'| + 1,\ \sigma' = \sigma^{k+}$$

$$[\![V]\!]_t\ \sigma\ \tau\ E \triangleq E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![V]\!]_V) \qquad [\![\mu\alpha_i.c]\!]_t\ \sigma\ \tau\ E \triangleq [\![c]\!]_c\ \sigma^+\ \tau[E]_E$$

$$[\![E]\!]_e\ \sigma\ \tau\ t \triangleq t\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![E]\!]_E) \qquad [\![\tilde{\mu}x_i.c]\!]_e\ \sigma\ \tau\ t \triangleq [\![c]\!]_c\ \sigma^+\ \tau[t]_t$$

$$[\![c\tau]\!]_l\ \sigma\ \tau' \triangleq [\![c]\!]_c\ \sigma'\ \tau'(\uparrow^\sigma [\![\tau]\!]_\tau) \qquad [\![\langle t \parallel e\rangle]\!]_c\ \sigma\ \tau \triangleq [\![e]\!]_e\ \sigma\ \tau\ (\uparrow^\sigma [\![t]\!]_t)$$
$$\text{where } k = |\tau'|,\ \sigma' = \sigma^{k+}$$

$$[\![\tau_0[x_i := t]]\!]_\tau \triangleq [\![\tau_0]\!]_\tau[[\![t]\!]_t]_t \qquad [\![\varepsilon]\!]_\tau \triangleq \varepsilon \qquad [\![\tau_0[\alpha_i := E]]\!]_\tau \triangleq [\![\tau_0]\!]_\tau[[\![E]\!]_E]_E$$

**Fig. 7.** Call-by-need translation of terms

2. Continuations/terms that are passed with a store are always consistent with it, *i.e.* they do not need to be lifted and their types match the store type.

Let us spend a few lines to explain the definitions of two cases, namely $[\![\tilde{\mu}[x_i].\langle x \parallel F\rangle \tau']\!]_E$ and $[\![x_i]\!]_V$. In the untyped translation, we have:

$$[\![\tilde{\mu}[x].\langle x \parallel F\rangle \tau']\!]_E\ \tau\ V \triangleq V\ \tau[x :=\uparrow^t V][\![\tau']\!]_\tau\ [\![F]\!]_F$$
$$[\![x]\!]_V\ \tau[x := t]\tau'\ F \triangleq t\ \tau\ (\lambda\tau\lambda V.V\ \tau[x :=\uparrow^t V]\tau'\ F)$$

Let us first focus on the $[\![\tilde{\mu}[x_i].\langle x \parallel F\rangle \tau']\!]_E$. In the named version, its translation is a term which waits for a store $\tau$ and a value $V$, then forms a store looking like $\tau[x := [\![V]\!]]\tau'$ and passes it to $V$ with the continuation $[\![F]\!]_F$. Now, when using de Bruijn levels, the continuation is somehow expecting a fragment $\tau_0$ of the store $\tau$ that is passed, together with a coercion $\sigma$ witnessing that $\tau$ is an extension of $\tau_0$ (let us loosely write $\sigma : \tau <: \tau_0$ to simplify our explanation). Therefore, only the value $V$ has its de Bruijn levels consistent with $\tau$, $[\![F]\!]_F$ and $[\![\tau']\!]_\tau$ will have to be updated. In details, $[\![\tau']\!]_\tau$ was expecting $\tau_0[\uparrow^t V]$, we thus update it with $\sigma^+$ which witnesses $\tau[\uparrow^t V] <: \tau_0[\uparrow^t V]$. On the other hand, $F$ was waiting for $\tau_0[\uparrow^t V]\tau'$, we thus need to update it with $\sigma^{k+}$, where $k = |\tau'| + 1$. Finally, we

$$
\begin{array}{l|l}
\begin{array}{lcl}
[\![\Gamma \vdash_e e : T^{\perp\!\!\!\perp}]\!] & \triangleq & \vdash [\![e]\!]_e : [\![\Gamma]\!]_\Gamma \rhd_e T \\
[\![\Gamma \vdash_t t : T]\!] & \triangleq & \vdash [\![t]\!]_t : [\![\Gamma]\!]_\Gamma \rhd_t T \\
[\![\Gamma \vdash_E E : T^{\perp\!\!\!\perp}]\!] & \triangleq & \vdash [\![E]\!]_E : [\![\Gamma]\!]_\Gamma \rhd_E T \\
[\![\Gamma \vdash_V V : T]\!] & \triangleq & \vdash [\![V]\!]_V : [\![\Gamma]\!]_\Gamma \rhd_V T \\
[\![\Gamma \vdash_F F : T^{\perp\!\!\!\perp}]\!] & \triangleq & \vdash [\![F]\!]_F : [\![\Gamma]\!]_\Gamma \rhd_F T
\end{array}
&
\begin{array}{lcl}
[\![\Gamma \vdash_v v : T]\!] & \triangleq & \vdash [\![v]\!]_v : [\![\Gamma]\!]_\Gamma \rhd_v T \\
[\![\Gamma \vdash_c c]\!] & \triangleq & \vdash [\![c]\!]_c : [\![\Gamma]\!]_\Gamma \rhd_c \bot \\
[\![\Gamma \vdash_l l]\!] & \triangleq & \vdash [\![l]\!]_l : [\![\Gamma]\!]_\Gamma \rhd_c \bot \\
[\![\Gamma \vdash_\tau \tau : \Gamma']\!] & \triangleq & \vdash [\![\tau]\!]_\tau : [\![\Gamma]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma
\end{array}
\end{array}
$$

$$
[\![\varepsilon]\!]_\Gamma \triangleq \varepsilon \qquad [\![\Gamma, x_i : T]\!]_\Gamma \triangleq [\![\Gamma]\!]_\Gamma, T \qquad [\![\Gamma, \alpha_i : T^{\perp\!\!\!\perp}]\!]_\Gamma \triangleq [\![\Gamma]\!]_\Gamma, T^{\perp\!\!\!\perp}
$$

$$
\begin{array}{l|l}
\begin{array}{ll}
\Upsilon \rhd_c T & \triangleq \forall Y <: \Upsilon. Y \to \bot \\
\Upsilon \rhd_e T & \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_t T) \to \bot \\
\Upsilon \rhd_t T & \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_E T) \to \bot \\
\Upsilon \rhd_E T & \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_V T) \to \bot
\end{array}
&
\begin{array}{ll}
\Upsilon \rhd_V T & \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_F T) \to \bot \\
\Upsilon \rhd_F T & \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_v T) \to \bot \\
\Upsilon \rhd_v X & \triangleq X \\
\Upsilon \rhd_v T \to U & \triangleq \\
\multicolumn{2}{l}{\quad \forall Y <: \Upsilon. Y \to (Y \rhd_t T) \to (Y \rhd_E U) \to \bot}
\end{array}
\end{array}
$$

**Fig. 8.** Call-by-need translation of judgments and types

need to give to $V$ a coercion witnessing the extension of $\tau$ into $\tau[\uparrow^t V]_t[\![\tau']\!]_\tau$, that is to say $\mathtt{shift}^k_{|\tau|}$. In the end, we obtain the following definition:

$$[\![\tilde{\mu}[x_i].\langle x_i \parallel F\rangle\tau']\!]_E\ \sigma\ \tau\ V \triangleq V\ \mathtt{id}_{|\tau|}\ (\tau[\uparrow^t V]_t(\uparrow^s [\![\tau']\!]_\tau))\ (\uparrow^{\sigma'} [\![F]\!]_F)$$

As for $[\![x_i]\!]_V$, it is a term waiting for a coercion $\sigma$ and a store $\tau$, which it will split at $[\![\sigma]\!](i)$ as $\tau_0, t, \tau_1$ to execute the term $t$ with its prefix $\tau_0$ (with which it is already consistent) and a continuation inlining the translation of $\tilde{\mu}[x_i].\langle x_i \parallel F\rangle\tau'$.

The translation of terms is given in Figure 7, where we assume that for each constant $\boldsymbol{k}$ of type $T$ (resp. co-constant $\boldsymbol{\kappa}$ of type $T^{\perp\!\!\!\perp}$) of the source system, we have a constant of type $T$ in the signature of the target language that we also denote by $\boldsymbol{k}$ (resp. $\boldsymbol{\kappa}$ of type $T \to \bot$).

**Translation of types** Regarding the translation of types, it follows exactly the intuition we presented in Section 3.1, so that we mostly already said everything about it. To summarize the construction, we start by embedding the types and typing contexts of the source calculus thanks to the $\iota$ function. A typing context $\Gamma = x_1 : T_1, \dots x_n : T_n$ is then translated into the store type $[\![\Gamma]\!]_\Gamma = T_1, \dots, T_n$. This allows us to translate a sequent, for instance $\Gamma \vdash_t t : T$, into a judgment $\vdash [\![t]\!]_t : [\![\Gamma]\!]_\Gamma \rhd_t T$. The type $[\![\Gamma]\!]_\Gamma \rhd_t T$ can be understood as the types of terms translated at level $t$, which are waiting for any store extending $[\![\Gamma]\!]_\Gamma$ and a continuation at the inferior level (that is $E$) for the same base type $T$:

$$[\![\Gamma]\!]_\Gamma \rhd_t T = \forall Y <: [\![\Gamma]\!]_\Gamma. Y \to (Y \rhd_E T) \to \bot$$

It is worth noting that the translation $- \rhd_t -$ is defined internally in system $F_\Upsilon$, which allows in particular the recursive definition $Y \rhd_E T$ to makes sense (since

$$
\begin{array}{rcl}
[\![\lambda x_i.t]\!]_V\ \sigma\ \tau\ u\ E & \triangleq & [\![t]\!]_t\ \sigma^+\ \tau[u]\ \uparrow^{\mathtt{shift}^1_{|\tau|}} E \\
[\![t \cdot E]\!]_E\ \sigma\ \tau\ v & \triangleq & v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![t]\!]_t)\ (\uparrow^\sigma [\![E]\!]_E) \\
[\![x_i]\!]_t\ \sigma\ \tau\ E & \triangleq & \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\ x\ (\mathtt{shift}^{|\delta_1|+1}_{|\delta_0|})\ \tau\ E \\
[\![\tilde{\mu} x_i.c]\!]_e\ \sigma\ \tau\ t & \triangleq & [\![c]\!]_c\ \sigma^+\ \tau[t]_t
\end{array}
$$

(a) Translation of terms (excerpt)

$$
\begin{array}{l}
[\![\Gamma \vdash_V V : T]\!] \triangleq \vdash\ [\![V]\!]_V : [\![\Gamma]\!]_\Gamma \rhd_V T \\
\Upsilon \rhd_V T \to U \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_t T) \to (Y \rhd_E U) \to \bot
\end{array}
$$

(b) Translation of types and judgments (excerpt)

**Fig. 9.** Call-by-name continuation-and-environment-passing style translation

$Y$ is at the moment an interleaving of types in $\Gamma$ and second-order variables that are not the images of types in the source calculus). As we already explained, the different levels $e, t, E, V, F, v$ of translation reflect the dynamics of the (context-free) reduction system of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, that is to say the different syntactic categories which are examined successively during the reduction[17].

The resulting translation is given in Figure 8.

**Correctness** We are finally equipped to state the main theorem of this paper, that is the correctness of the translation with respect to types.

**Theorem 10.** *The translation is well-typed, i.e.:*

1. *If* $\Gamma \vdash_v v : T$ *then* $[\![\Gamma \vdash_v v : T]\!]$
2. *If* $\Gamma \vdash_F F : T^{\perp\!\!\!\perp}$ *then* $[\![\Gamma \vdash_F F : T^{\perp\!\!\!\perp}]\!]$
3. *If* $\Gamma \vdash_V V : T$ *then* $[\![\Gamma \vdash_V V : T]\!]$
4. *If* $\Gamma \vdash_E E : T^{\perp\!\!\!\perp}$ *then* $[\![\Gamma \vdash_E E : T^{\perp\!\!\!\perp}]\!]$
5. *If* $\Gamma \vdash_t t : T$ *then* $[\![\Gamma \vdash_t t : T]\!]$
6. *If* $\Gamma \vdash_e e : T^{\perp\!\!\!\perp}$ *then* $[\![\Gamma \vdash_e e : T^{\perp\!\!\!\perp}]\!]$
7. *If* $\Gamma \vdash_c c$ *then* $[\![\Gamma \vdash_c c]\!]$
8. *If* $\Gamma \vdash_l l$ *then* $[\![\Gamma \vdash_l l]\!]$
9. *If* $\Gamma \vdash_\tau \tau$ *then* $[\![\Gamma \vdash_\tau \tau : \Gamma']\!]$

*Proof.* The proof is done by induction on typing derivations, the interesting parts of the different cases corresponding to the previous lemmas. The complete proof is given in Appendix E, together with a few auxiliary lemmas.

### 4.2 A typed call-by-name translation for the $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments

To emphasize that $F_\Upsilon$ is a generic target calculus for typed continuation-and-environment-passing style translations, we give the example of a call-by-name translation for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments (see Figure 2). We spare the reader from the redefinition of its reduction rules and type system

[17] The context-free reduction rules are given in Appendix C.

using de Bruijn levels, which is fully deducible from the ones of the $\overline{\lambda}_{[lv\tau\star]}$-calculus when considering only the levels $e, t, E, V$ and typing terms and contexts at the appropriate level.

Similarly, we do not wish to enter into too many details about the translation of terms and types. We follow the same process as for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, by refining the dynamic of the calculus into a context-free abstract machine (see Appendix C.2). This machine only has four level of alternation, as reflected by the syntax, and so does both translations of terms and types. The definition of the translation almost comes for free modulo the careful treatment of de Bruijns levels. Most of the definitions are identical (or simpler) than in the call-by-need case, the main difference lying in the fact that in call-by-name, terms remains unevaluated in the store, thus avoiding the need for an extra layer of alternation to handle their (shared) evaluation. We give a few cases of these translations in Fig. 16 (for the full translation see Appendix F). Once again, we have:

**Theorem 11.** *The translation is well-typed.*

## 5 Conclusion and perspectives

### 5.1 Conclusion

In this paper, we presented a variant of the $\overline{\lambda}_{[lv\tau\star]}$-calculus with de Bruijn levels. We showed how this calculus could be equipped with a type system in order to define a typed continuation-and-environment-passing style translation to a target language that we called System $F_{\Upsilon}$. We believe that the principles guiding the typing of the translation emphasized its computational content, whose three main ingredients are the following:

1. a continuation-passing style translation,
2. an environment-passing style translation,
3. a Kripke forcing-like manner of typing the extensibility of the environment.

The latter is particularly highlighted by the use of de Bruijn levels, since levels are shifted using coercions when extending the store, these coercions thus giving a computational content to the subtyping relation (*i.e.* to store extension). To enhance the fact that our technique is generic, we illustrated it on several flavors of calculi with environment, namely the call-by-need $\overline{\lambda}_{[lv\tau\star]}$-calculus and the call-by-name and call-by-value $\bar{\lambda}\mu\tilde{\mu}$-calculi with environments.

### 5.2 About environments and forcing

Actually, the connection between (Kripke) forcing and the environment-passing style translation does not come as a surprise. Indeed, the translation on types logically accounts for the compilation of the calculus with environments to a calculus without environments. In the realm of functional programming, memory states are given a meaning through the state monad. For instance, the monadic translation of an arrow enriches it with a state $S$:

$$[\![T \to U]\!] \triangleq S \times T \to S \times U$$

In particular, the result of a function may depend on the current state. If one observes precisely the realizability interpretation in [32], it is very similar to the definition of truth and falsity values: for a type $T$, its interpretation is roughly of the shape $T \times \tau$. It is folklore that the state monad can be categorically interpreted by means of presheaves construction [37,29]. Interestingly, Kripke models are a particular case of presheaves semantics [33]. Cohen forcing construction is also interpreted in terms of presheaves [27], and this interpretation scales to type theory [19,18]. Therefore, the state monad and the forcing translation were already known to be connected. Last but not least, the analysis of Cohen forcing in the framework of Krivine classical realizability [23,30] relies on an extension of Krivine abstract machine with a cell (which contains the forcing condition). In short, our typed environment-passing style translation is just another observation of the connection between forcing translations and explicit environments as a side-effect.

### 5.3 Further work

This work naturally raises the question of studying the logical strength of the translations. Indeed, we were initially interested in a computational translation of classical arithmetic with dependent choice to System F. Both systems are indeed known to be logically equivalent, and such a translation has been defined in the converse direction by Blot [7]. In a recent paper [31], the second author defined dLPA$^\omega$, a sequent calculus for classical arithmetic with dependent choice, which is a reformulation based on the $\overline{\lambda}_{[lv\tau\star]}$-calculus of the first author's dPA$^\omega$ system [17]. While the present CPS translation can be tailored for dLPA$^\omega$, as such we still do not know whether it can be typed using only System F. On the one hand, System $F_\Upsilon$ seems to be stronger than Systems F or F$_{<:}$ in that it allows a restricted form of dependent types: the second-order quantification range over vectors of arbitrary size. On the other hand, it is probably weaker than a higher order calculus with unrestricted dependencies in types, like the calculus of constructions (which is logically as strong as F$_\omega$). Yet, it might also be the case that a clever analysis of the translations could lead to a bound on the size of the store extension at each step. This would offer a way to remove this dependency and to embed the target language into System F.

Furthermore, following Kesner's work, several recent papers have been using intersection type system to characterize normalizing by-need terms [20,6]. Even though these calculi are not classical, it might be interesting to adapt her approach to our framework. Specifically, we have the intuition that intersection types could be an alternative to our subtyping relation in the target language of the translation.

Last, we would like to study whether our construction is compatible with more complex source type systems. We indeed showed in this paper that it was generic for different simply-typed calculi, the next step would then naturally be to get a construction that is suitable for other type systems such as Systems F or F$^\omega$.

## References


1. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 363–376. ICFP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628154, http://doi.acm.org/10.1145/2628136.2628154
2. Accattoli, B., Barras, B.: Environments and the Complexity of Abstract Machines. In: The 19th International Symposium on Principles and Practice of Declarative Programming. Namur, Belgium (Oct 2017). https://doi.org/10.1145/3131851.3131855, https://hal.archives-ouvertes.fr/hal-01675358
3. Appel, A.W.: Compiling with Continuations. Cambridge University Press, New York, NY, USA (1992)
4. Ariola, Z., Felleisen, M.: The call-by-need lambda calculus. J. Funct. Program. **7**(3), 265–301 (1993). https://doi.org/10.1017/S0956796897002724
5. Ariola, Z.M., Downen, P., Herbelin, H., Nakata, K., Saurin, A.: Classical call-by-need sequent calculi: The unity of semantic artifacts. In: Schrijvers, T., Thiemann, P. (eds.) Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings. pp. 32–46. Lecture Notes in Computer Science, Springer (2012). https://doi.org/10.1007/978-3-642-29822-6
6. Balabonski, T., Barenbaum, P., Bonelli, E., Kesner, D.: Foundations of strong call by need. Proc. ACM Program. Lang. **1**(ICFP), 20:1–20:29 (Aug 2017). https://doi.org/10.1145/3110264, http://doi.acm.org/10.1145/3110264
7. Blot, V.: An interpretation of system f through bar recursion. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–12 (June 2017). https://doi.org/10.1109/LICS.2017.8005066
8. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5), 381 – 392 (1972). https://doi.org/http://dx.doi.org/10.1016/1385-7258(72)90034-0
9. Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system F with subtyping, pp. 750–770. Springer Berlin Heidelberg, Berlin, Heidelberg (1991), http://dx.doi.org/10.1007/3-540-54415-1_73
10. Crégut, P.: Strongly reducing variants of the krivine abstract machine. Higher-Order and Symbolic Computation **20**(3), 209–230 (Sep 2007). https://doi.org/10.1007/s10990-007-9015-z
11. Curien, P.L., Herbelin, H.: The duality of computation. In: Proceedings of ICFP 2000. pp. 233–243. SIGPLAN Notices 35(9), ACM (2000). https://doi.org/10.1145/351240.351262
12. Danvy, O., Millikin, K., Munk, J., Zerny, I.: Defunctionalized Interpreters for Call-by-Need Evaluation, pp. 240–256. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
13. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the lambda-calculus. In: 3rd Working Conference on the Formal Description of Programming Concepts (1986)
14. Felleisen, M., Sabry, A.: Continuations in programming practice: Introduction and survey (1999), https://www.cs.indiana.edu/~sabry/papers/continuations.ps, manuscript

15. Felleisen, M., Friedman, D.P., Kohlbecker, E., Duba, B.: A syntactic theory of sequential control. Theor. Comput. Sci. **52**(3), 205–237 (Jun 1987). https://doi.org/10.1016/0304-3975(87)90109-5, http://dx.doi.org/10.1016/0304-3975(87)90109-5
16. Griffin, T.G.: A formulae-as-type notion of control. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 47–58. POPL '90, ACM, New York, NY, USA (1990). https://doi.org/10.1145/96709.96714, http://doi.acm.org/10.1145/96709.96714
17. Herbelin, H.: 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. pp. 365–374. IEEE Computer Society (2012). https://doi.org/10.1109/LICS.2012.47, http://dx.doi.org/10.1109/LICS.2012.47
18. Jaber, G., Lewertowski, G., Pédrot, P.M., Sozeau, M., Tabareau, N.: The definitional side of the forcing. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 367–376. LICS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2933575.2935320
19. Jaber, G., Tabareau, N., Sozeau, M.: Extending type theory with forcing. In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science. pp. 395–404. LICS '12, IEEE Computer Society, Washington, DC, USA (2012). https://doi.org/10.1109/LICS.2012.49
20. Kesner, D.: Reasoning About Call-by-need by Means of Types, pp. 424–441. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
21. Kripke, S.A.: Semantical considerations on modal logic. Acta Philosophica Fennica **16**(1963), 83–94 (1963)
22. Krivine, J.L.: A call-by-name lambda-calculus machine. In: Higher Order and Symbolic Computation (2004)
23. Krivine, J.L.: Realizability algebras: a program to well order r. Logical Methods in Computer Science **7**(3) (2011)
24. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal **6**(4), 308–320 (1964). https://doi.org/10.1093/comjnl/6.4.308
25. Lang, F.: Explaining the lazy krivine machine using explicit substitution and addresses. Higher-Order and Symbolic Computation **20**(3), 257–270 (Sep 2007). https://doi.org/10.1007/s10990-007-9013-1
26. Leroy, X.: The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA (1990)
27. MacLane, S., Moerdijk, I.: Sheaves in Geometry and Logic. Springer (1992). https://doi.org/10.1007/978-1-4612-0927-0
28. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. J. Funct. Program. **8**(3), 275–317 (1998). https://doi.org/10.1017/S0956796898003037
29. Melliès, P.A.: Local States in String Diagrams, pp. 334–348. Springer International Publishing, Cham (2014)
30. Miquel, A.: Forcing as a program transformation. In: LICS. pp. 197–206. IEEE Computer Society (2011)
31. Miquey, E.: A sequent calculus with dependent types for classical arithmetic. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 720–729. LICS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209199, http://doi.acm.org/10.1145/3209108.3209199

32. Miquey, É., Herbelin, H.: Realizability Interpretation and Normalization of Typed Call-by-Need $\lambda$-calculus With Control. In: FOSSACS 18 - 21st International Conference on Foundations of Software Science and Computation Structures. Thessalonique, Greece (Apr 2018). https://doi.org/10.1007/978-3-319-89366-2˙15
33. Moerdijk, I., van Oosten, J.: Topos theory (2007), `http://www.staff.science.uu.nl/~ooste110/syllabi/toposmoeder.pdf`
34. Murthy, C.: Extracting constructive content from classical proofs. Ph.D. thesis, Cornell University (1990)
35. Okasaki, C., Lee, P., Tarditi, D.: Call-by-need and continuation-passing style. Lisp and Symbolic Computation **7**(1), 57–82 (1994). https://doi.org/10.1007/BF01019945
36. Parigot, M.: Proofs of strong normalisation for second order classical natural deduction. J. Symb. Log. **62**(4), 1461–1479 (1997)
37. Plotkin, G., Power, J.: Notions of Computation Determine Monads, pp. 342–356. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
38. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975). https://doi.org/10.1016/0304-3975(75)90017-1
39. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. **7**(3), 231–264 (May 1997). https://doi.org/10.1017/S0956796897002712
40. Sussman, G.J., Steele, Jr., G.L.: An interpreter for extended lambda calculus. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1975)

$$
\begin{array}{rcl}
\overline{t}\,\overline{u} \star \pi \star \tau & \to_{\mathtt{c}} & \overline{t} \star \overline{u} \cdot \pi \star \tau \\
\lambda x.\overline{t} \star u \cdot \pi \star \tau & \to_{\beta} & \overline{t} \star \pi \star \tau[x := \overline{u}] \\
x \star \pi \star \tau[x := \overline{t}]\tau' & \to_{\mathtt{s}} & \overline{t}^{\alpha} \star \pi \star \tau[x := \overline{t}]\tau'
\end{array}
$$

a. Milner Abstract Machine

$$
\begin{array}{rcl}
\langle t \parallel \tilde{\mu}x.c\rangle\tau & \to & c\tau[x := t] \\
\langle \mu\alpha.c \parallel E\rangle\tau & \to & c\tau[\alpha := E] \\
\langle V \parallel \alpha\rangle\tau[\alpha := E]\tau' & \to & \langle V \parallel E\rangle\tau[\alpha := E]\tau' \\
\langle x \parallel E\rangle\tau[x := t]\tau' & \to & \langle t \parallel E\rangle\tau[x := t]\tau' \\
\langle \lambda x.t \parallel u \cdot E\rangle\tau & \to & \big\langle u \,\big\|\, \tilde{\mu}x.\langle t \parallel E\rangle\big\rangle\tau
\end{array}
$$

b. The $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments

**Fig. 10.** Milner Abstract Machine and $\bar{\lambda}\mu\tilde{\mu}$-calculus

# A Simulations of Milner Abstract Machines with sequent calculi

## A.1 The MAM and the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments

It is quite obvious that the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments allows us to faithfully simulate reductions of the MAM (which we recall in Fig. 10). We first define the following compilation function from states of the MAM to closures of the $\bar{\lambda}\mu\tilde{\mu}$-calculus:

$$[\![t \star \pi \star \tau]\!] \triangleq \langle [\![t]\!]_t \parallel [\![\pi]\!]_\pi \rangle [\![\tau]\!]_\tau$$

with:

$$
\begin{array}{rcl}
[\![x]\!]_t & \triangleq & x \\
[\![\lambda x.t]\!]_t & \triangleq & \lambda x.[\![t]\!]_t \\
[\![tu]\!]_t & \triangleq & \mu\alpha.\langle [\![t]\!]_t \parallel [\![u]\!]_t \cdot \alpha\rangle
\end{array}
\qquad
\begin{array}{rcl}
[\![u \cdot \pi]\!]_\pi & \triangleq & [\![u]\!]_t \cdot [\![\pi]\!]_\pi \\
[\![\varepsilon]\!]_\pi & \triangleq & \kappa \\
[\![\tau[x := t]]\!]_\tau & \triangleq & [\![\tau]\!]_\tau[x := [\![t]\!]_t] \\
[\![\varepsilon]\!]_\tau & \triangleq & \varepsilon
\end{array}
$$

where $\boldsymbol{\kappa}$ is a fixed co-constant materializing the end of the execution.

It is then quite easy to verify that reductions of the MAM are preserved through the compilation process (modulo the fact that we consider terms of the $\bar{\lambda}\mu\tilde{\mu}$-calculus up to $\alpha$-conversion). Formally, to avoid considering the contexts stored in the environment, we first define a substitution function for co-variables. We write $(c\tau)\{\tau'\}$ for the closure $c\tau$ in which all the co-variables $\alpha$ bound in $\tau'$ are recursively substituted by the terms to which they are bound:

$$
\begin{array}{c}
(c\tau)\{\tau'[x := t]\} \triangleq (c\tau)\{\tau'\} \\
(c\tau)\{\tau'[\alpha := E]\} \triangleq ((c\tau)[E/\alpha])\{\tau'\} \\
(c\tau)\{\varepsilon\} \triangleq c\tau
\end{array}
$$

We then say that two closures $c\tau$ and $c'\tau'$ are equal up to co-variables substitutions, which we denote by $c\tau \equiv c'\tau'$, whenever $(c\tau)\{\tau\} = (c'\tau')\{\tau'\}$.

**Proposition 1 (MAM simulation).** *If $\mathcal{S}, \mathcal{S}'$ are two states of the MAM such that $\mathcal{S} \xrightarrow{1} \mathcal{S}'$, then there exists a closure $c'\tau'$ such that we have $[\![\mathcal{S}]\!] \xrightarrow{+} c'\tau'$ and $(c'\tau') \equiv [\![\mathcal{S}']\!]$.*

*Proof.* Trivial induction on reduction rules of the MAM.

In other words, we showed that a variant of the $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments where catchable contexts would be immediately substituted instead of being stored is exactly simulating the MAM through the compilation function.

### A.2 The MAD and the $\overline{\lambda}_{[lv\tau\star]}$-calculus

Similarly, we can prove that the $\overline{\lambda}_{[lv\tau\star]}$-calculus allows us to simulate reductions of the MAD. The compilation function from states of the MAD to closures of the $\overline{\lambda}_{[lv\tau\star]}$-calculus is almost the same, except that we know need to take the dump into account. As we explain in Section 2.4, the dump is somehow inlined in the $\overline{\lambda}_{[lv\tau\star]}$-calculus through the binder $\tilde{\mu}[x].\langle x \parallel F\rangle\tau'$. Following this intuition, the definition of the translation is almost direct:

$$[\![t \star \pi \star \tau \star D]\!] \triangleq \langle [\![t]\!]_t \parallel [\![\pi]\!]_\pi^{[\![D]\!]_D} \rangle [\![\tau]\!]_\tau$$

with:

$$\begin{array}{rcl} [\![x]\!]_t & \triangleq & x \\ [\![\lambda x.t]\!]_t & \triangleq & \lambda x.[\![t]\!]_t \\ [\![tu]\!]_t & \triangleq & \mu\alpha.\langle [\![t]\!]_t \parallel [\![u]\!]_t \cdot \alpha\rangle \end{array} \quad \left| \quad \begin{array}{rcl} [\![u \cdot \pi]\!]_\pi^e & \triangleq & [\![u]\!]_t \cdot [\![\pi]\!]_\pi^e \\ [\![\varepsilon]\!]_\pi^e & \triangleq & e \end{array} \right. \quad \left| \quad \begin{array}{rcl} [\![\tau[x := t]]\!]_\tau & \triangleq & [\![\tau]\!]_\tau[x := [\![t]\!]_t] \\ [\![\varepsilon]\!]_\tau & \triangleq & \varepsilon \end{array}$$

$$[\![(x, \pi, \tau) :: D]\!]_D \triangleq \tilde{\mu}[x].\langle x \parallel [\![\pi]\!]_\pi^{[\![D]\!]_D} \rangle [\![\tau]\!]_\tau \quad \Big| \quad [\![\varepsilon]\!]_D \triangleq \boldsymbol{\kappa}$$

Once again, it is straightforward to check that:

**Proposition 2 (MAD simulation).** *If $\mathcal{S}, \mathcal{S}'$ are two states of the MAD such that $\mathcal{S} \xrightarrow{1} \mathcal{S}'$, then there exists a closure $c'\tau'$ such that we have $[\![\mathcal{S}]\!] \xrightarrow{+} c'\tau'$ and $(c'\tau') \equiv [\![\mathcal{S}']\!]$.*

*Proof.* Trivial induction on reduction rules of the MAD.

| | | | |
|---|---|---|---|
| **Strong values** | $v ::= \lambda x.t \mid \boldsymbol{k}$ | **Environments** | $\tau ::= \varepsilon \mid \tau[x := t] \mid \tau[\alpha := E]$ |
| **Weak values** | $V ::= v \mid x$ | **Commands** | $c ::= \langle t \parallel e \rangle$ |
| **Terms** | $t, u ::= V \mid \mu\alpha.c$ | **Closures** | $l ::= c\tau$ |

| | | |
|---|---|---|
| **Forcing contexts** | $F$ | $::= t \cdot E \mid \boldsymbol{\kappa}$ |
| **Catchable contexts** | $E$ | $::= F \mid \alpha \mid \tilde{\mu}[x].\langle x \parallel F \rangle\tau$ |
| **Evaluation contexts** | $e$ | $::= E \mid \tilde{\mu}x.c$ |

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

**Fig. 11.** The $\overline{\lambda}_{[lv\tau\star]}$-calculus

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

## B.1 Definitions

We recall here the definition of Ariola *et al.*'s $\overline{\lambda}_{[lv\tau\star]}$-calculus [5]. The syntax and reduction rules are given in Fig. 11, while the type system, defined in [32], is given in Fig. 12. Finally, Ariola *et al.*'s original untyped CPS translation is given in Fig. 13.

## B.2 The necessity of $\alpha$-renaming

The original presentation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus deeply relies on the assumption that names of variable are unique and thus on the possibility of performing $\alpha$-conversion on-the-fly. Consider for instance a command formed by a term of the shape $t = \mu\alpha.\langle\, u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\,\rangle$ and a context of the shape $e = \tilde{\mu}x.\langle\, x \parallel F\,\rangle$. Such a command is perfectly typable (if $u$ and $F$ are) in the type system introduced in [32], however, reducing this command (without $\alpha$-conversion) would loop forever because of the auto-reference $[x := x]$ in the environment:

$$
\begin{aligned}
\Big\langle \mu\alpha.\big\langle u \,\big\|\, \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle \,\Big\|\, \tilde{\mu}x.\langle x \parallel F\rangle \Big\rangle &\to \langle x \parallel F\rangle[x := \mu\alpha.\big\langle u \,\big\|\, \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle] \\
&\to \Big\langle \mu\alpha.\big\langle u \,\big\|\, \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle \,\Big\|\, \tilde{\mu}[x].\langle x \parallel F\rangle \Big\rangle \\
&\to \big\langle u \,\big\|\, \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle] \\
&\to \langle x \parallel \alpha\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle, x := u] \\
&\to \big\langle x \,\big\|\, \tilde{\mu}[x].\langle x \parallel F\rangle\big\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle, x := u] \\
&\to \langle x \parallel F\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle, x := u, x := x] \to \dots
\end{aligned}
$$

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

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

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

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

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

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

**Fig. 12.** Typing rules for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

While a simple $\alpha$-conversion of one of the $x$ binding solves the problem, this becomes much more subtle to handle through a CPS translation without renaming (as the one in Figure 13 originally defined in [5]). Indeed, since "different" variables named $x$ (that is variables which are bound by different binders) are translated independently (*e.g.* $[\![ \langle t \parallel e \rangle ]\!]$ is defined from $[\![e]\!]$ and $[\![t]\!]$), there is no hope to perform $\alpha$-conversion on the fly during the translation. Thus, the problem becomes unsolvable after the translation, and the problem of renaming should be tackled together with the definition of the translation of terms. For instance, through this translation, the same closure is again a program that will loop forever:

$$\begin{aligned}
[\![c\varepsilon]\!] &= [\![e]\!]_e\ \varepsilon\ [\![t]\!]_t = [\![\tilde{\mu}x.\langle x \parallel F\rangle]\!]_e\ \varepsilon\ [\![t]\!]_t \\
&= [\![\langle x \parallel F\rangle]\!]_c\ [x := [\![t]\!]_t] \\
&= [\![x]\!]_x\ [x := [\![t]\!]_t]\ [\![F]\!]_F \\
&= [\![\mu\alpha.\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle]\!]_t\ \varepsilon\ (\lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F) \\
&= [\![\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle]\!]_t\ [\alpha := \lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F] \\
&= [\![\tilde{\mu}x.\langle x \parallel \alpha\rangle]\!]_e\ [\alpha := \lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F]\ [\![u]\!]_t \\
&= [\![\langle x \parallel \alpha\rangle]\!]_c\ [\alpha := \lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F, x := [\![u]\!]_t] \\
&= [\![\alpha]\!]_E\ [\alpha := \lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F, x := [\![u]\!]_t]\ [\![x]\!]_V \\
&= (\lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V])\ [\alpha := \lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F, x := [\![u]\!]_t]\ [\![x]\!]_V \\
&\to [\![x]\!]_V\ [\alpha := \lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\ [\![F]\!]_F, x := [\![u]\!]_t, x := [\![x]\!]_t]
\end{aligned}$$

$$
\begin{array}{rcl}
[\![c\ \tau]\!]_l\ \tau_0 & \triangleq & [\![c]\!]_c\ \tau_0\tau' \\
[\![\langle t \parallel e\rangle]\!]_c\ \tau & \triangleq & [\![e]\!]_e\ \tau\ [\![t]\!]_t \\
\\
[\![\varepsilon]\!]_\tau & \triangleq & \varepsilon \\
[\![\tau'[x := t]]\!]_\tau & \triangleq & [\![\tau']\!]_\tau[x := [\![t]\!]_t] \\
[\![\tau'[\alpha := E]]\!]_\tau & \triangleq & [\![\tau']\!]_\tau[x := [\![E]\!]_E] \\
\\
[\![E]\!]_e\ \tau\ t & \triangleq & t\ \tau\ [\![E]\!]_E \\
[\![\tilde{\mu}x.c]\!]_e\ \tau\ t & \triangleq & [\![c]\!]_c\ \tau[x := t] \\
\\
[\![V]\!]_t\ \tau\ E & \triangleq & E\ \tau\ [\![V]\!]_V \\
[\![\mu\alpha.c]\!]_t\ \tau\ E & \triangleq & [\![c]\!]_c\ \tau[\alpha := E] \\
[\![\alpha]\!]_E\ \tau[\alpha := E]\tau'\ V & \triangleq & E\ \tau[\alpha := E]\tau'\ V \\
[\![\tilde{\mu}[x].\langle x \parallel F\rangle\tau']\!]_E\ \tau\ V & \triangleq & V\ \tau[x := \lambda\tau E.E\,\tau\,V][\![\tau']\!]_\tau\ [\![F]\!]_F \\
\\
[\![v]\!]_V\ \tau\ F & \triangleq & F\ \tau\ [\![v]\!]_v \\
[\![x]\!]_V\ \tau[x := t]\tau'\ F & \triangleq & t\ \tau\ (\lambda\tau\lambda V.V\ \tau[x := \lambda\tau E.E\,\tau\,V]\tau'\ F) \\
\\
[\![\boldsymbol{\kappa}]\!]_F & \triangleq & \boldsymbol{\kappa} \\
[\![t \cdot E]\!]_F\ \tau\ v & \triangleq & v\ \tau\ [\![t]\!]_t\ [\![E]\!]_E \\
\\
[\![\boldsymbol{k}]\!]_v & \triangleq & \boldsymbol{k} \\
[\![\lambda x.t]\!]_v\ \tau\,u\,E & \triangleq & [\![t]\!]_t\ \ \tau[x := u]\ E
\end{array}
$$

**Fig. 13.** Ariola *et al.* untyped CPS translation

Observe that as the translation is defined modulo administrative reduction, the first equations indeed are equalities, and that when the reduction is performed, the two "different" $x$ are not bound anymore. Thus, there is no way to achieve any kind of $\alpha$-conversion to prevent the formation of the cyclic reference $[x := [\![x]\!]_V]$. This is why we need either to be able to perform $\alpha$-conversion while executing the translation of a command, assuming that we can find a smooth way to do it, or to explicitly handle the renaming.

In order to ensure the correctness of our translation, we address the problem at the source in the $\overline{\lambda}_{[lv\tau\star]}$, using de Bruijn levels. As we observed in the previous example, the issue arises when adding a binding $[x := ...]$ in an environment that already contained a variable $x$. We thus need to ensure the uniqueness of names within the environment. A simple solution consists in renaming the variables bound in the environment by the position at which they occur in the environment, which is obviously unique. Before presenting formally the corresponding system and the adapted translation, let us reduce the same example using this idea. We use a mixed notation for names, writing $x$ when a variable is bound by a $\lambda$ or a $\tilde{\mu}$, and $x_i$ (where $i$ is the relevant information) when it refers to a position in the environment. The same reduction is now safe if we replace stored

variables by their de Bruijn levels:

$$\Big\langle \mu\alpha.\big\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle \Big\| \tilde{\mu}x.\langle x \parallel F\rangle\Big\rangle \to \langle x_0 \parallel F\rangle[^0\mu\alpha.\big\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle]$$
$$\to \quad \Big\langle \mu\alpha.\big\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle \Big\| \tilde{\mu}[x].\langle x \parallel F\rangle\Big\rangle \quad \to \quad \big\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha_0\rangle\big\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle]$$
$$\to \quad \langle x_1 \parallel \alpha_0\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle, {}^1u] \quad \to \quad \big\langle x_1 \parallel \tilde{\mu}[x].\langle x \parallel F\rangle\big\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle, {}^1u]$$
$$\to \quad \langle x_1 \parallel F\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle, {}^1u, {}^2x_1] \quad \to \quad \big\langle u \parallel \tilde{\mu}[x].\langle x \parallel F\rangle[^2x_1]\big\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle]$$

where the exponents $^0, {}^1, ...$ to number the cells are only there to ease the readability.

Another solution would have consisted in defining the translation using an explicit renaming function. In broad lines, the translation of terms (resp. contexts, closures, etc.) should be of the form $[\![-]\!]_t^\sigma$ where $\sigma$ is a substitution used to rename variables. To compact the notations, we write $[{}^x_m|^\alpha_\gamma|...]$ for the renaming substitution $[x := m, \alpha := \gamma, ...]$, where we adopt the convention that the most recent binding is on written on the right. As a binding $[x := n]$ overwrites any former binding $[x := m]$, we write $[{}^\alpha_\gamma|^x_n]$ instead of $[{}^x_m|^\alpha_\gamma|^x_n]$. Using this trick, the translation of the former command would be (where $m$ and $n$ are fresh names generated during the translation):

$$\begin{aligned}
[\![c\varepsilon]\!]^\varepsilon &= [\![e]\!]^\varepsilon_e\ \varepsilon\ [\![t]\!]^\varepsilon_t = [\![\tilde{\mu}x.\langle x \parallel F\rangle]\!]^\varepsilon_e\ \varepsilon\ [\![t]\!]^\varepsilon_t \\
&= [\![\langle x \parallel F\rangle]\!]_c^{[^x_m]}\ [m := [\![t]\!]_t] \\
&= [\![x]\!]_t^{[^x_m]}\ [m := [\![t]\!]^\varepsilon_t]\ [\![F]\!]_F^{[^x_m]} \\
&= [\![\mu\alpha.\big\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle]\!]_t^{[^x_m]}\ \varepsilon\ (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}) \\
&= [\![\big\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\big\rangle]\!]_t^{[^x_m|^\alpha_\gamma]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}] \\
&= [\![\tilde{\mu}x.\langle x \parallel \alpha\rangle]\!]_e^{[^x_m|^\alpha_\gamma]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}]\ [\![u]\!]_t^{[^x_m|^\alpha_\gamma]} \\
&= [\![\langle x \parallel \alpha\rangle]\!]_c^{[x:=m,\alpha:=\gamma,x:=n]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}, n := [\![u]\!]_t^{[^x_m|^\alpha_\gamma]}] \\
&= [\![\alpha]\!]_E^{[^x_m|^\alpha_\gamma|^x_n]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}, n := [\![u]\!]_t^{[^x_m|^\alpha_\gamma]}][\![x]\!]_V^{[^x_m|^\alpha_\gamma|^x_n]} \\
&= (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V])\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}, n := [\![u]\!]_t^{[^x_m|^\alpha_\gamma]}]\ [\![x]\!]_V^{[^\alpha_\gamma|^x_n]} \\
&\to [\![x]\!]_V^{[^\alpha_\gamma|^x_n]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}, n := [\![u]\!]_t^{[^x_m|^\alpha_\gamma]}, m := [\![x]\!]_t^{[^\alpha_\gamma|^x_n]}] \\
&= [\![x]\!]_V^{[^\alpha_\gamma|^x_n]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ [\![F]\!]_F^{[^x_m]}, n := [\![u]\!]_t^{[^x_m|^\alpha_\gamma]}, m := [\![x]\!]_t^{[^\alpha_\gamma|^x_n]}]
\end{aligned}$$

We observe that in the end, the variable $m$ is bound to the variable $n$, which is now correct. While this method has the benefit of avoiding the reformulation of the source calculus with de Bruijn levels, it has the flaw of hiding a part of the computational content related to the renaming process (that is Kripke forcing).

### B.3 De Bruijn levels

We give here the formal definition of the lifted term $\uparrow_n^{+i} t$, the term $t$ where all the free variables $x_j$ with $j > n$ (resp. $\alpha_j$) have been replaced by $x_{j+i}$ (resp. $\alpha_{i+j}$). Given $i, n$ two natural numbers, we define:

$$
\begin{array}{ll}
\uparrow_n^{+i}(c\tau) & \triangleq (\uparrow_n^{+i} c)(\uparrow_n^{+i} \tau) \\
\uparrow_n^{+i}(\langle t \parallel e \rangle) & \triangleq \langle \uparrow_n^{+i} t \parallel \uparrow_n^{+i} e \rangle \\
 & \\
\uparrow_n^{+i}(\boldsymbol{\kappa}) & \triangleq \boldsymbol{\kappa} \\
\uparrow_n^{+i}(t \cdot E) & \triangleq (\uparrow_n^{+i} t) \cdot (\uparrow_n^{+i} E) \\
\uparrow_n^{+i}(\alpha_j) & \triangleq \alpha_j \qquad (\text{if } j < n) \\
\uparrow_n^{+i}(\alpha_j) & \triangleq \alpha_{j+i} \qquad (\text{if } j \geq n) \\
\uparrow_n^{+i}(\tilde{\mu}[x_j].\langle x_j \parallel F \rangle \tau) & \triangleq \tilde{\mu}[\uparrow_n^{+i} x_j].(\uparrow_n^{+i} \langle x_j \parallel F \rangle \tau) \\
\uparrow_n^{+i}(\tilde{\mu} x_j.c) & \triangleq \tilde{\mu}(\uparrow_n^{+i} x_j).(\uparrow_n^{+i} c) \\
 & \\
\uparrow_n^{+i} \varepsilon & \triangleq \varepsilon \\
\uparrow_n^{+i}(\tau[x_j := t]) & \triangleq \uparrow_n^{+i}(\tau)([\uparrow_n^{+i} x_j :=\uparrow_n^{+i} t] \\
\uparrow_n^{+i}(\tau[\alpha_j := E]) & \triangleq \uparrow_n^{+i}(\tau[\uparrow_n^{+i} \alpha_j :=\uparrow_n^{+i} E] \\
 & \\
\uparrow_n^{+i}(\boldsymbol{k}) & \triangleq \boldsymbol{k} \\
\uparrow_n^{+i}(\lambda x_j.t) & \triangleq \lambda(\uparrow_n^{+i} x_j).(\uparrow_n^{+i} t) \\
\uparrow_n^{+i}(x_j) & \triangleq x_j \qquad (\text{if } j < n) \\
\uparrow_n^{+i}(x_j) & \triangleq x_{j+i} \qquad (\text{if } j \geq n) \\
\uparrow_n^{+i}(\mu \alpha_j.c) & \triangleq \mu(\uparrow_n^{+i} \alpha_j).(\uparrow_n^{+i} c)
\end{array}
$$

# C Context-free abstract machines

## C.1 The named context-free abstract machine for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

$$
\begin{array}{rcl}
\langle t \parallel \tilde{\mu}x.c\rangle_e\tau & \to & c_e\tau[x := t] \\
\langle t \parallel E\rangle_e\tau & \to & \langle t \parallel E\rangle_t\tau \\
\langle \mu\alpha.c \parallel E\rangle_t\tau & \to & c_e\tau[\alpha := E] \\
\langle V \parallel E\rangle_t\tau & \to & \langle V \parallel E\rangle_E\tau \\
\langle V \parallel \alpha\rangle_E\tau[\alpha := E]\tau' & \to & \langle V \parallel E\rangle_E\tau[\alpha := E]\tau' \\
\left\langle V \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle\tau'\right\rangle_E\tau & \to & \langle V \parallel F\rangle_V\tau[x := V]\tau' \\
\langle V \parallel F\rangle_E\tau & \to & \langle V \parallel F\rangle_V\tau \\
\langle x \parallel F\rangle_V\tau[x := t]\tau' & \to & \left\langle t \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle\tau'\right\rangle\tau \\
\langle v \parallel E\rangle_V\tau & \to & \langle v \parallel F\rangle_V\tau \\
\langle v \parallel u \cdot E\rangle_F\tau & \to & \langle v \parallel u \cdot E\rangle_v\tau \\
\langle \lambda x.t \parallel u \cdot E\rangle_v\tau & \to & \left\langle u \,\middle\|\, \tilde{\mu}x.\langle t \parallel E\rangle\right\rangle_e\tau
\end{array}
$$

**Fig. 14.** Context-free abstract machine for the $\overline{\lambda}_{[lv\tau\star]}$-calculus [5]

## C.2 Context-free abstract machine for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments

$$
\begin{array}{rcl}
\langle t \parallel \tilde{\mu}x_i.c\rangle_e\tau & \to & c[x_n/x_i]_e\tau[x_n := t] \quad \text{with } |\tau| = n \\
\langle t \parallel E\rangle_e\tau & \to & \langle t \parallel E\rangle_t\tau \\
\langle \mu\alpha_i.c \parallel E\rangle_t\tau & \to & c[\alpha.n/\alpha_i]_e\tau[\alpha_n := E] \text{ with } |\tau| = n \\
\langle x_n \parallel E\rangle_t\tau & \to & \langle \tau(n) \parallel E\rangle_t \\
\langle V \parallel E\rangle_t\tau & \to & \langle V \parallel E\rangle_E\tau \\
\langle V \parallel \alpha_n\rangle_E\tau & \to & \langle V \parallel \tau(n)\rangle_E\tau \\
\langle V \parallel u \cdot E\rangle_E\tau & \to & \langle V \parallel u \cdot E\rangle_V\tau \\
\langle \lambda x.t \parallel u \cdot E\rangle_v\tau & \to & \left\langle u \,\middle\|\, \tilde{\mu}x.\langle t \parallel E\rangle\right\rangle_e\tau
\end{array}
$$

**Fig. 15.** Context-free abstract machine for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments

## D Properties of System $F_{\Upsilon}$

We give here the properties of System $F_{\Upsilon}$with their proofs.

**Properties of system $F_{\Upsilon}$** First, it is clear that the type system is compatible with a weakening rule:

**Lemma 12 (Weakening)** *The following rule is admissible:*

$$\frac{\Gamma \vdash t : A \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash t : A} \ (\Gamma_w)$$

*Proof.* Easy induction on typing derivations. In the case of second-order quantification, we might need to rename the second-order variable $X$ if it occurs in $\Gamma'$ and not in $\Gamma$.

We can verify type safety with respect to reductions:

**Proposition 2 (Subject reduction).** For any context $\Gamma$, any type $A$ and any terms $t, t'$, if $\Gamma \vdash t : A$ and $t \to t'$, then $\Gamma \vdash t' : A$.

*Proof.* The proof is standard and does not bring much information. We start by proving the following statements for safe substitutions:

1. If $\Gamma, x : A, \Gamma' \vdash t : B$ and $\Gamma \vdash u : A$, then $\Gamma, \Gamma' \vdash t[u/x] : B$.
2. If $\Gamma, \delta : \Upsilon_0 \rhd_\tau \Upsilon_1, \Gamma' \vdash t : B$ and $\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon_1$, then $\Gamma, \Gamma' \vdash t[\tau/\delta] : B$.
3. If $\Gamma, s : Y <: \Upsilon, \Gamma' \vdash t : B$ and $\Gamma \vdash \sigma : \Upsilon' <: \Upsilon$, then $\Gamma, \Gamma'[\Upsilon'/Y] \vdash t[\sigma/s] : B[\Upsilon'/Y]$.

Each of the three statements is proved together with the similar statements for typing judgments for stores and coercions by mutual induction on typing derivations. The proof of subject reduction is then direct by induction on reduction rules using the previous statements to conclude.

**Normalization** It should be clear to the reader that system $F_{\Upsilon}$ could be defined through a more expressive language including a form a dependent types and a second-order quantification. In particular, it could be expressed within many type theories or proof assistants, which especially implies that typed terms are normalizing. Once again, we could make this statement more formal but as we already dispose of a proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we prefer to focus on the details of the different translations.

**Coercions** We recall the definition of coercions in normal forms:

**Normal forms** $\sigma ::= \varepsilon \mid \sigma^+ \mid \ \Uparrow\sigma$

We define the composition function $-\odot-$ between coercions in normal form by:

$$\begin{array}{ll} \sigma_1^+ \odot \sigma_0^+ \triangleq (\sigma_1 \odot \sigma_0)^+ & \Uparrow\sigma_1 \odot \sigma_0 \triangleq \Uparrow(\sigma_1 \odot \sigma_0) \\ \sigma_1^+ \odot \Uparrow\sigma_0 \triangleq \Uparrow(\sigma_1 \odot \sigma_0) & \varepsilon \odot \sigma_0 = \sigma_1^+ \odot \varepsilon \triangleq \varepsilon \end{array}$$

We can verify that this function is sound with respect to the typing rule for composing coercions:

**Lemma 3 (Composition of normal forms).** If $\sigma, \sigma'$ are coercions in normal forms such that $\vdash \sigma : \mathcal{T} <: \mathcal{T}'$ and $\vdash \sigma' : \mathcal{T}' <: \mathcal{T}''$, then $\vdash \sigma' \odot \sigma : \mathcal{T} <: \mathcal{T}''$.

*Proof.* Direct by structural induction on $\sigma'$.

We define *computable coercions* as being coercions of the shape:

**Computable coercions** $\qquad \sigma ::= \varepsilon \mid \sigma^+ \mid \quad \Uparrow\sigma \mid \sigma' \circ \sigma$

**Proposition 4 (Computing normal forms).** For any computable $\sigma$, if $\Gamma \vdash \sigma : \mathcal{T}' <: \mathcal{T}$ then there exists $\sigma_n$ in normal form such that $\Gamma \vdash \sigma_n : \mathcal{T}' <: \mathcal{T}$.

*Proof.* By induction on typing derivations, using the previous lemma for the $(<:_\circ)$ -rule.

It is worth noting that for any $\sigma, \mathcal{T}, \mathcal{T}'$, if $\vdash \sigma : \mathcal{T}' <: \mathcal{T}$, then $\sigma$ is necessarily computable. We then deduce as a corollary of the previous proposition:

**Corollary 5.** If $\vdash \sigma : \mathcal{T}' <: \mathcal{T}$, then $|\mathcal{T}| \leq |\mathcal{T}'|$.

*Proof.* Using the previous proposition, we can reduce this to the case of $\sigma$ in normal forms. The proof then proceed by easy structural induction on $\sigma$.

If $\sigma$ is a coercion in normal form (if it is computable we first reduce it to a computation in normal form), we define its domain $\mathrm{dom}(\sigma)$ and its codomain $\mathtt{codom}(\sigma)$ by:

$$\begin{array}{ccc|ccc|ccc} \mathrm{dom}(\varepsilon) & \triangleq & 0 & \mathrm{dom}(\sigma^+) & \triangleq & \mathrm{dom}(\sigma)+1 & \mathrm{dom}(\Uparrow\sigma) & \triangleq & \mathrm{dom}(\sigma) \\ \mathtt{codom}(\varepsilon) & \triangleq & 0 & \mathtt{codom}(\sigma^+) & \triangleq & \mathtt{codom}(\sigma)+1 & \mathtt{codom}(\Uparrow\sigma) & \triangleq & \mathtt{codom}(\sigma)+1 \end{array}$$

We then associate to $\sigma$ the partial function $[\![\sigma]\!]$ from $[0, \mathrm{dom}(\sigma)]$ to $[0, \mathtt{codom}(\sigma)]$ defined by:

$$\begin{array}{l|l} [\![\varepsilon]\!] \triangleq \{0 \mapsto 0\} \\ [\![\sigma^+]\!] \triangleq [\![\sigma]\!] \cup \{\mathrm{dom}(\sigma) \mapsto \mathtt{codom}(\sigma)\} & [\![\Uparrow\sigma]\!] \triangleq \begin{cases} n < \mathrm{dom}(\sigma) \mapsto [\![\sigma]\!](n) \\ n = \mathrm{dom}(\sigma) \mapsto [\![\sigma]\!](n)+1 \end{cases} \end{array}$$

**Proposition 6 (Associated function).** If $\sigma$ is in normal form and s.t. $\vdash \sigma : \mathcal{T}' <: \mathcal{T}$, then:

1. $\mathrm{dom}(\sigma) = |\Upsilon|$
2. $\mathtt{codom}(\sigma) = |\Upsilon'|$
3. $\forall n < |\Upsilon|, \Upsilon'(\llbracket\sigma\rrbracket(n)) = \Upsilon(n)$
4. $\llbracket\sigma\rrbracket(|\Upsilon|) = |\Upsilon'|$

*Proof.* The first two items are proved by a straightforward induction on typing derivations. The third and forth items are then proved together, again by induction on typing derivations: the case $(<:_\varepsilon)$ is trivial; while for $(<:_+)$ and $(<:_\Uparrow)$ it suffices to unfold the definition, using the first items to connect $|\Upsilon|$ and $\mathrm{dom}(\sigma)$.

Let us define $\mathtt{id}_0 = \emptyset$ and $\mathtt{id}_{n+1} = (\mathtt{id}_n)^+$, we have that:

**Proposition 13 (Subtyping)** *In the empty context, the subtyping relation $<:$ is an order relation on store types.*

1. *For any $\Upsilon$, the rule* $\dfrac{}{\vdash \mathtt{id}_{|\Upsilon|} : \Upsilon <: \Upsilon}\ {}^{(<:_{\mathtt{id}})}$ *is admissible.*
2. *If* $\vdash \sigma : \Upsilon <: \Upsilon'$ *and* $\vdash \sigma' : \Upsilon' <: \Upsilon''$, *then* $\vdash \sigma' \circ \sigma : \Upsilon <: \Upsilon''$.
3. *If* $\vdash \sigma : \Upsilon <: \Upsilon'$ *and* $\vdash \sigma' : \Upsilon' <: \Upsilon$, *then* $\Upsilon = \Upsilon'$.

*Proof.* Straightforward using the previous lemma to reduce it to the case of coercions in normal forms. The first two items are straightforward. As for the third one, it is a direct consequence of Proposition 6, since $\llbracket\sigma\rrbracket$ and $\llbracket\sigma'\rrbracket$ are two strictly monotone functions from $[0, |\Upsilon|]$ to itself (by Corollary 5 we have $|\Upsilon'| = |\Upsilon|$), they are necessarily the identity. Equivalently, we could have seen that necessarily $\sigma$ is of the shape $\sigma_0^+$ (and so is $\sigma'$), so that $\Upsilon' = \Upsilon'_0, T$ and $\Upsilon = \Upsilon_0, T$, from which we can conclude by an easy induction.

Last, we introduce the following shorthands:

$$\sigma^{0+} \triangleq \sigma \qquad \sigma^{(k+1)+} \triangleq (\sigma^+)^{k+} \qquad \Uparrow^0 \sigma \triangleq \sigma \qquad \Uparrow^{k+1} \sigma \triangleq \Uparrow^k (\Uparrow \sigma) \qquad \mathtt{shift}_n^k \triangleq \Uparrow^k \mathtt{id}_n$$

Indeed, we have that:

**Lemma 7 (Shifts).** For any $\Upsilon_0, \Upsilon'_0, \Upsilon_1$, writing $n = |\Upsilon_0|$ and $k = |\Upsilon_1|$ we have:

$$\frac{\Gamma \vdash \sigma : \Upsilon'_0 <: \Upsilon_0}{\Gamma \vdash \sigma^{k+} : \Upsilon'_0 \Upsilon_1 <: \Upsilon_0 \Upsilon_1} \qquad \frac{}{\Gamma \vdash \mathtt{shift}_n^k : \Upsilon_0 \Upsilon_1 <: \Upsilon_0}$$

*Proof.* Both proofs are easy inductions on $|\Upsilon_1|$.

**Lifting terms and stores** Finally, we conclude this section by showing how terms and stores can be lifted using coercions for their types to remain consistent while extended stores are passed in the translations. First, we show that the bounded quantification can be composed with a subtyping relation witnessed by a coercion $\sigma$, by precomposing terms with $\sigma$. Given a coercion $\sigma$ of type $\Upsilon' <: \Upsilon$ and a term $t$ whose type is of the shape $\forall Y <: \Upsilon.A$, we define:

$$(\uparrow^\sigma t) \triangleq \lambda s.t\ (s \circ \sigma)$$

**Lemma 8.** The following rule is admissible:

$$\frac{\Gamma \vdash t : \forall Y <: \Upsilon.A \quad \Gamma \vdash \sigma : \Upsilon' <: \Upsilon}{\Gamma \vdash (\uparrow^\sigma t) : \forall Y <: \Upsilon'.A}\ (\uparrow^\sigma)$$

*Proof.* We assume that $Y$ is fresh with respect to $FV(\Gamma)$, otherwise it suffices to rename it. Unfolding the definition of $\uparrow^\sigma t$, we can derive:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t : \forall Y <: \Upsilon_0.A}{\Gamma, s : Y <: \Upsilon_1 \vdash t : \forall Y <: \Upsilon_0.A}\,(\Gamma_w) \quad \dfrac{\Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0 \quad \dfrac{}{\Gamma, s : Y <: \Upsilon_1 \vdash s : X <: \Upsilon_1}\,(<:_{\mathtt{ax}})}{\Gamma, s : Y <: \Upsilon_1 \vdash s \circ \sigma : Y <: \Upsilon_0}\,(<:_\circ)}{\Gamma, s : Y <: \Upsilon_1 \vdash t\ (s \circ \sigma) : A}\,(\forall_E) \quad Y \notin FV(\Gamma)}{\Gamma \vdash \lambda s.t\ (s \circ \sigma) : \forall Y <: \Upsilon_1.A}\,(\forall_I)$$

where we use Lemma 12 to weaken $\Gamma, \sigma : X <: \Upsilon_1$.

When the translations $\Upsilon \rhd_t T$ and $\Upsilon \rhd_E T$ are of the shape $\forall Y <: \Upsilon....$, this definition can be scaled to stores by setting:

$$\uparrow^\sigma(\varepsilon) \triangleq \varepsilon \qquad \uparrow^\sigma([t]_t\tau) \triangleq [\uparrow^\sigma t]_t(\uparrow^{\sigma^+}\tau) \qquad \uparrow^\sigma([t]_E\tau) \triangleq [\uparrow^\sigma t]_E(\uparrow^{\sigma^+}\tau)$$

Observe that in a store $[t]_t[u]_t\tau$, $t$ is lifted with $\sigma$ while $u$ is lifted with $\sigma^+$ (and so on recursively). This is due to the fact that if $\sigma$ is of type $\Upsilon' <: \Upsilon$ and the store of type $\Upsilon \rhd_\tau T, U, ...$, the term $t$ is then of type $\Upsilon \rhd_t T$ and can be lifted with $\sigma$. In turns, $u$ is of some type $\Upsilon, T \rhd_t U$ and thus requires to be lifted with a coercion of type $\Upsilon', T <: \Upsilon, T$, that is to say $\sigma^+$. More generally, we deduce from the former lemma the following corollary that will be crucial when typing the translation of terms:

**Corollary 9.** If $\Upsilon \rhd_t T$ (resp. $\Upsilon \rhd_E T$) is of the shape $\forall X <: \Upsilon.F(X, A)$ (resp. $\forall X <: \Upsilon.G(X, T)$), then the following rules is admissible:

$$\frac{\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^\sigma \tau) : \Upsilon_1 \rhd_\tau \Upsilon}$$

*Proof.* The former lemma directly gives us that

$$\frac{\Gamma \vdash t : \Upsilon_0 \rhd_t T \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^\sigma t) : \Upsilon_1 \rhd_t T} \qquad \frac{\Gamma \vdash t : \Upsilon_0 \rhd_E T \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^\sigma t) : \Upsilon_1 \rhd_E T}$$

The proof then simply proceeds by induction on the structure of $\tau$.

## E Proof of well-typedness

We give here the complete proof of the correction of the translation of terms with respect to types. We begin by making a few observations that will be useful in the proof of the main theorem.

First of all, it is easy to check that the rules for forming stores and witnessing extensions are safe through the translation:

**Lemma 14 (Store formation)** *The following rules are admissible:*

$$\frac{\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon \quad \Gamma \vdash t : \Upsilon_0, \Upsilon \rhd_t T}{\Gamma \vdash \tau[t]_t : \Upsilon_0 \rhd_\tau \Upsilon, T}\ (\tau[t]) \qquad \frac{\Gamma \vdash \sigma : \Upsilon <: [\![\Gamma_0]\!]_\Gamma}{\Gamma \vdash \sigma^+ : \Upsilon, T <: [\![\Gamma_0, T]\!]_\Gamma}$$

*The same holds for* $\Gamma \vdash E : \Upsilon_0, \Upsilon \rhd_E T$ *and* $\Gamma \vdash \tau[E]_E : \Upsilon_0 \rhd_\tau \Upsilon, T^{\perp\!\!\!\perp}$.

*Proof.* Straightforward typing derivations. For instance, for the left-hand side we have:

$$\frac{\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon \quad \dfrac{\Gamma \vdash t : \Upsilon_0, \Upsilon \rhd_t T}{\Gamma \vdash [t]_t : \Upsilon_0, \Upsilon \rhd_\tau T}\ (\tau_t)}{\Gamma \vdash \tau[t]_t : \Upsilon_0 \rhd_\tau \Upsilon, T}\ (\tau\tau')$$

Since the translation of types satisfies the hypothesis of Corollary 9, we deduce from the previous lemma that:

**Corollary 15** *For any level $o$ of the hierarchy $e, t, E, V, F, v$, the following rule are admissible:*

$$\frac{\Gamma \vdash t : \Upsilon_0 \rhd_o T \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^\sigma t) : \Upsilon_1 \rhd_o T}\ (\uparrow^\sigma) \qquad \frac{\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^\sigma \tau) : \Upsilon_1 \rhd_\tau \Upsilon}$$

**Lemma 16 (Lifting values)** *The following rule is admissible:*

$$\frac{\Gamma \vdash V : \Upsilon \rhd_V T}{\Gamma \vdash \uparrow^t V : \Upsilon \rhd_t T}\ (\uparrow)$$

*Proof.* We can derive (weakening contexts on-the-fly to ease readability):

$$\dfrac{\dfrac{\boldsymbol{\Pi_E} \quad \dfrac{\Gamma \vdash V : \Upsilon \rhd_V T \quad \overline{s : Y <: \Upsilon \vdash s : Y <: \Upsilon}\ (<:_{\mathtt{ax}})}{\Gamma, s : Y <: \Upsilon \vdash \uparrow^s V : Y \rhd_V T}\ (\uparrow^\sigma)}{\Gamma, s : Y <: \Upsilon, \delta : Y, E : \Upsilon \rhd_E T \vdash E\ \mathtt{id}_{|\delta|}\ \delta\ (\uparrow^s V) : \perp}\ (@)}{\Gamma \vdash \lambda s \delta E.E\ \mathtt{id}_{|\delta|}\ \delta\ (\uparrow^s V) : \Upsilon \rhd_t T}\ (\lambda)$$

where we used Corollary 15 and $\Pi_E$ is the following derivation:

$$\dfrac{\dfrac{\overline{E : \Upsilon \rhd_E T \vdash E : Y \rhd_E T}\ (\mathrm{Ax}) \quad \overline{\delta : Y \vdash \mathtt{id}_{|\delta|} : Y <: Y}\ (<:_{\mathtt{id}})}{\delta : Y, E : \Upsilon \rhd_E T \vdash E\ \mathtt{id}_{|\delta|} : Y \to Y \rhd_V T \to \perp}\ (\forall_E) \quad \overline{\delta : Y \vdash \delta : Y}\ (\mathrm{Ax})}{\delta : Y, E : \Upsilon \rhd_E T \vdash E\ \mathtt{id}_{|\delta|}\ \delta : Y \rhd_V T \to \perp}\ (@)$$

Observe that we implicitly use the fact that since $\delta : Y$, by definition $|\delta| = |Y|$.

We are finally equipped to prove the main theorem:

**Theorem 10.** The translation is well-typed, i.e.:

1. If $\Gamma \vdash_v v : T$ then $[\![\Gamma \vdash_v v : T]\!]$
2. If $\Gamma \vdash_F F : T^{\perp\!\!\!\perp}$ then $[\![\Gamma \vdash_F F : T^{\perp\!\!\!\perp}]\!]$
3. If $\Gamma \vdash_V V : T$ then $[\![\Gamma \vdash_V V : T]\!]$
4. If $\Gamma \vdash_E E : T^{\perp\!\!\!\perp}$ then $[\![\Gamma \vdash_E E : T^{\perp\!\!\!\perp}]\!]$
5. If $\Gamma \vdash_t t : T$ then $[\![\Gamma \vdash_t t : T]\!]$
6. If $\Gamma \vdash_e e : T^{\perp\!\!\!\perp}$ then $[\![\Gamma \vdash_e e : T^{\perp\!\!\!\perp}]\!]$
7. If $\Gamma \vdash_c c$ then $[\![\Gamma \vdash_c c]\!]$
8. If $\Gamma \vdash_l l$ then $[\![\Gamma \vdash_l l]\!]$
9. If $\Gamma \vdash_\tau \tau$ then $[\![\Gamma \vdash_\tau \tau : \Gamma']\!]$

*Proof.* We reason by induction over typing derivations. We (ab)use of Lemma 12 to make the derivations more compact by systematically weakening contexts as soon as possible, and compact the first $(\forall_I)$ and $(\lambda)$ rules in one rule.

**1. Strong values $[\![\boldsymbol{k}]\!]_v = \boldsymbol{k}$**, which has the desired type by hypothesis.

- **Case** $\lambda x_i.t$. In the source language, we have:

$$\frac{\Gamma, x_i : T \vdash_t t : U \quad |\Gamma| = i}{\Gamma \vdash_v \lambda x_i.t : T \to U}$$

Hence, we get by induction a proof $\Pi_t$ of $[\![t]\!]_t : [\![\Gamma, x_i : T]\!] \rhd_t U$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\vdash [\![t]\!]_t : \forall Y' <: [\![\Gamma, x_i : T]\!].Y' \to Y' \rhd_E U \to \bot}\ \boldsymbol{\Pi_\sigma}}{s : Y <: [\![\Gamma]\!] \vdash [\![t]\!]_t\ s^+ : (Y,T) \to (Y,T) \rhd_E U \to \bot}\ (\forall_E)\quad \boldsymbol{\Pi_\tau}}{\sigma : Y <: [\![\Gamma]\!], \tau : Y, u : Y \rhd_t T \vdash [\![t]\!]_t\ s^+\ \tau[u] : (Y,T) \rhd_E U \to \bot}\ (@)\quad \boldsymbol{\Pi_E}}{s : Y <: [\![\Gamma]\!], \tau : Y, u : Y \rhd_t T, E : Y \rhd_E U \vdash [\![t]\!]_t\ s^+\ \tau[u]\ \uparrow^{\mathtt{shift}^1_{|\tau|}} E : \bot}\ (@)}{\vdash \lambda s \tau u E.[\![t]\!]_t\ s^+\ \tau[u]\ \uparrow^{\mathtt{shift}^1_{|\tau|}} E : \forall Y <: [\![\Gamma]\!].Y \to Y \rhd_t T \to Y \rhd_E U \to \bot}\ (\lambda)$$

where:

- $\Pi_E$ is a proof of $E : Y \rhd_E U \vdash \uparrow^{\mathtt{shift}^1_{|\tau|}} E : (Y,T) \rhd_E U$ (derivable according to Corollary 15);
- $\Pi_\tau$ is a proof of $\tau : Y, u : Y \rhd_t T; \vdash \tau[u] : Y, T$ (derivable according to Lemma 14);
- $\Pi_\sigma$ is simply:

$$\frac{\dfrac{}{\sigma : Y <: [\![\Gamma]\!] \vdash \sigma : Y <: [\![\Gamma]\!]}\ (<:_{\mathtt{ax}})}{s : Y <: [\![\Gamma]\!] \vdash s^+ : Y, T <: [\![\Gamma, x_i : T]\!]}\ (<:_+)$$

**2. Forcing contexts**

- **Case** $[\![\boldsymbol{\kappa}]\!]_F$. $[\![\boldsymbol{\kappa}]\!]_F = \boldsymbol{\kappa}$, which has the desired type by hypothesis.
- **Case** $[\![t.E]\!]_F$. In the source language, we have:

$$\frac{\Gamma \vdash_t t : T \quad \Gamma \vdash_E E : U^{\perp\!\!\!\perp}}{\Gamma \vdash_F t \cdot E : (T \to U)^{\perp\!\!\!\perp}}$$

Therefore, we have by induction hypothesis that $\vdash [\![t]\!]_t : [\![\Gamma]\!]_\Gamma \rhd_t T$ and $\vdash [\![E]\!]_t : [\![\Gamma]\!]_\Gamma \rhd_E U$, so that we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : Y \rhd_v T \to U; \vdash v : \forall Y' <: Y : Y' \to Y' \rhd_t T \to Y' \rhd_E U \to \bot}\,(\mathrm{Ax})\quad \boldsymbol{\Pi_\sigma}}{\tau : Y, v : Y \rhd_v T \to U; \vdash v\ \mathtt{id}_{|\tau|} : Y \to Y \rhd_t T \to Y \rhd_E U \to \bot}\,(\forall_I)\quad \boldsymbol{\Pi_\tau}}{\tau : Y, v : Y \rhd_v T \to U; \vdash v\ \mathtt{id}_{|\tau|}\ \tau : Y \rhd_t T \to Y \rhd_E U \to \bot}\,(@)\quad \boldsymbol{\Pi_t}}{\sigma : Y <: [\![\Gamma]\!], \tau : Y, v : Y \rhd_v T \to U \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![t]\!]_t) : Y \rhd_E U \to \bot}\,(@)\quad \boldsymbol{\Pi_E}}{\sigma : Y <: [\![\Gamma]\!], \tau : Y, v : Y \rhd_v T \to U \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![t]\!]_t)\ (\uparrow^\sigma [\![E]\!]_E) : \bot}\,(@)}{\vdash \lambda\sigma\tau v.v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![t]\!]_t)\ (\uparrow^\sigma [\![E]\!]_E) : \forall Y <: [\![\Gamma]\!]_\Gamma . Y \to Y \rhd_v T \to U \to \bot}\,(\lambda)$$

where:

- $\Pi_E$ is a proof of $\sigma : Y <: [\![\Gamma]\!] \vdash (\uparrow^\sigma [\![E]\!]_E) : Y \rhd_E U$, derived from the induction hypothesis for $t$ and Corollary 9;
- $\Pi_t$ is a proof of $\sigma : Y <: [\![\Gamma]\!] \vdash (\uparrow^\sigma [\![t]\!]_t) : Y \rhd_t T$, derived from the induction hypothesis for $E$ and Corollary 9;
- $\Pi_\tau$ is the axiom rule $\tau : Y \vdash \tau : Y$;
- $\Pi_\sigma$ is a proof of $\mathtt{id}_{|\tau|} : Y <: Y$ (Proposition 13).

## 3. Weak values

- **Case** $[\![v]\!]_V$. In the source language, we have:

$$\frac{\Gamma \vdash_v v : T}{\Gamma \vdash_V v : T}$$

Hence, we have by induction hypothesis that $\vdash [\![v]\!]_v : [\![\Gamma]\!]_\Gamma \rhd_v T$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{F : Y \rhd_F T \vdash F : \forall Y' <: Y . Y' \to Y' \rhd_v T \to \bot \quad \Pi_Y}{\sigma : Y <: [\![\Gamma]\!], F : Y \rhd_F T \vdash F\ \mathtt{id}_{|\tau|} : Y \to Y \rhd_v T \to \bot}\,(@) \quad \dfrac{}{\tau : Y; \vdash \tau : Y}}{\sigma : Y <: [\![\Gamma]\!], \tau : Y, F : Y \rhd_F T \vdash F\ \mathtt{id}_{|\tau|}\ \tau : Y \rhd_v T \to \bot}\,(@) \quad \Pi_v}{\sigma : Y <: [\![\Gamma]\!], \tau : Y, F : Y \rhd_F T \vdash F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![v]\!]_v) : \bot}\,(@)}{\vdash \lambda\sigma\tau F.F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![v]\!]_v) : \forall Y <: [\![\Gamma]\!] . Y \to Y \rhd_F T \to \bot}\,(\lambda)$$

where:

- $\Pi_v$ is a proof of $\sigma : Y <: [\![\Gamma]\!] \vdash (\uparrow^\sigma [\![v]\!]_v) : Y \rhd_v T$, derivable from the induction hypothesis and Corollary 9.
- $\Pi_\tau$ is the axiom rule $\tau : Y; \vdash \tau : Y$
- $\Pi_Y$ is a proof of $\mathtt{id}_{|\tau|} : Y <: Y$ (Proposition 13)

- **Case** $[\![x_i]\!]_V$. In the source language, we have:

$$\frac{\Gamma(i) = (x_i : T)}{\Gamma \vdash_V x_i : T}$$

so that $\Gamma$ is of the form $\Gamma_0, x_i : T, \Gamma_1$ with $i = |\Gamma_0|$. By definition, we have:

$$\begin{aligned}[\![x_i]\!]_V =& \lambda\sigma\tau F.\, \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1 \\ & \mathtt{in}\, x\ \mathtt{id}_{|\delta_0|}\ \delta_0\ (\lambda s\delta_0' V.V\ (\mathtt{shift}^p_{n+k})\ (\delta_0'[\uparrow^t V]_t \uparrow^s \delta_1)\ (\uparrow^{\sigma'} F))\end{aligned}$$

where $n = |\delta_0| = [\![\sigma]\!](i)$, $k = |\delta_0'| - n$, $p = |\delta_1| + 1$, $\sigma' = s^{p+}$ and $\uparrow^t V = \lambda s\delta E.E\ \mathtt{id}_{|\delta|}\ \delta\ (\uparrow^s V)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : Y_0 \rhd_t T \vdash x : Y_0 \rhd_t T}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash \mathtt{id}_{|\delta_0|} : Y_0 <: Y_0}\,(<:\mathtt{ax})}{x : Y_0 \rhd_t T \vdash x\ \mathtt{id}_{|\delta_0|} : Y_0 \to Y_0 \rhd_E T \to \bot}\,(\forall_E) \quad \dfrac{}{\delta_0 : Y_0 \vdash \delta_0 : Y_0}\,(\mathrm{Ax})}{\delta_0 : Y_0, x : Y_0 \rhd_t a \vdash t\ \mathtt{id}_{|\delta_0|}\ \delta_0 : Y_0 \rhd_E T \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{F : Y \rhd_F T, \delta_0 : Y_0, x : Y_0 \rhd_t a, \delta_1 : Y_0, T \rhd_\tau Y_1 \vdash x\ \mathtt{id}_{|\delta_0|}\ \delta_0\ E : \bot}\,(@) \quad \boldsymbol{\Pi_\sigma} \quad \boldsymbol{\Pi_\tau}}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y, F : Y \rhd_F T \vdash \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\, x\ \mathtt{id}_{|\delta_0|}\ \delta_0\ E : \bot}\,(\mathtt{split})}{\vdash \lambda\sigma\tau F.\, \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\, x\ \mathtt{id}_{|\delta_0|}\ \delta_0\ E : \forall Y <: [\![\Gamma]\!]_\Gamma.Y \to Y \rhd_F T \to \bot}\,(\lambda)$$

where:

- $\Pi_\sigma$ is the axiom rule:

$$\dfrac{}{\sigma : Y <: [\![\Gamma_0]\!]_\Gamma, T, [\![\Gamma_1]\!]_\Gamma \vdash \sigma : Y_0, T, Y_1 <: ([\![\Gamma_0]\!]_\Gamma, n : T, [\![\Gamma_1]\!]_\Gamma)}\,(<:\mathtt{ax})$$

  remembering the fact that the notation $\sigma : Y <: [\![\Gamma]\!]_\Gamma$ hides an interleaving of vectors of second-order variables with the types in $[\![\Gamma]\!]_\Gamma$. In particular, $Y_0$ is again a notation for an interleaving between such vectors and $[\![\Gamma_0]\!]_\Gamma$, and besides, we have that $\sigma(|\Gamma_0|) = \sigma(i) = |Y_0| = n$.
- $\Pi_\tau$ is the axiom rule: $\dfrac{}{\tau : Y \vdash \tau : Y}\,(\tau\mathtt{ax})$
- $E = \lambda s\delta_0' V.V\ (\mathtt{shift}^p_{n+k})\ (\delta_0'[\uparrow^t V]_t \uparrow^s \delta_1)\ (\uparrow^{\sigma'} F)$ and $\Pi_E$ is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y_0' \rhd_V T; \vdash V : Y_0' \rhd_t T}\,(\mathrm{Ax}) \quad \dfrac{\text{Lemma 7}}{\Gamma, \vdash (\mathtt{shift}^p_{n+k}) : Y_0', T, Y_1 <: Y_0'}}{\Gamma, V : Y_0' \rhd_V T \vdash V\ (\mathtt{shift}^p_{n+k}) : (Y_0', T, Y_1) \to (Y_0', T, Y_1) \rhd_F T \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi_\tau'}}{\Gamma, s : Y_0' <: Y_0, V : Y_0' \rhd_V T \vdash V\ (\mathtt{shift}^p_{n+k})\ (\delta_0'[\uparrow^t V]_t \uparrow^{s^+} \delta_1) : (Y_0', T, Y_1) \rhd_F T \to \bot}\,(@) \quad \boldsymbol{\Pi_f}}{\Gamma, s : Y_0' <: Y_0, \delta_0' : Y_0', V : Y_0' \rhd_V T \vdash V\ (\mathtt{shift}^p_{n+k})\ (\delta_0'[\uparrow^t V]_t \uparrow^{s^+} \delta_1)\ (\uparrow^{\sigma'} F) : \bot}\,(@)}{\Gamma \vdash \lambda s\delta_0' V.V\ (\mathtt{shift}^p_{n+k})\ (\delta_0'[\uparrow^t V]_t \uparrow^{s^+} \delta_1)\ (\uparrow^{\sigma'} F) : Y_0 \rhd_E T}\,(\lambda)$$

  where $\Gamma = \delta_1 : Y_0, T) \rhd_\tau Y_1, F : (Y_0, T, Y_1) \rhd_F T$.

- $\Pi_F$ is the following proof, obtained by Corollary 15:

$$\dfrac{\dfrac{}{F : (Y_0, T, Y_1) \rhd_F T; \vdash F : (Y_0, T, Y_1) \rhd_F T}\,(\mathrm{Ax}) \quad \dfrac{\text{Lemma 7}}{s_0' : Y_0' <: Y_0 \vdash \sigma' : Y_0', T, Y_1 <: Y_0, T, Y_1}\,(\mathrm{Ax})}{F : (Y_0, T, Y_1) \rhd_F T; \sigma_1 : Y_0' <: Y_0^n \vdash (\uparrow^{\sigma'} F) : (Y_0', T, Y_1) \rhd_F T}\,(\uparrow^\sigma)$$

– $\Pi'_\tau$ is the following derivation, where we used Lemmas 14 and 16:

$$\dfrac{\dfrac{\dfrac{}{\delta'_0 : Y'_0 \vdash \delta'_0 : Y'_0}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{V : Y'_0 \rhd_V T \vdash V : Y'_0 \rhd_V T}\,(\mathrm{Ax})}{V : Y'_0 \rhd_V T \vdash \uparrow^t V : Y'_0 \rhd_t T}\,(\uparrow)}{Y'_0 <: Y_0, \delta'_0 : Y'_0, V : Y'_0 \rhd_V T \vdash \delta'_0[\uparrow^t V] : Y'_0, T}\,(\tau[t]) \quad \boldsymbol{\Pi}_{\delta_1}}{\delta_1 : (Y_0, T) \rhd_\tau Y_1, s : Y'_0 <: Y_0, \delta'_0 : Y'_0, V : Y'_0 \rhd_V T \vdash \delta'_0[\uparrow^t V]_t \uparrow^{s^+} \delta_1 : Y'_0, T, Y_1}\,(\tau_{<:})$$

– $\Pi_{\delta_1}$ is obtained by Corollary 15:

$$\dfrac{\dfrac{}{\delta_1 : (Y_0, T) \rhd_\tau Y_1 \vdash \delta_1 : (Y_0, T) \rhd_\tau Y_1}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{s : Y'_0 <: Y_0 \vdash s : Y'_0 <: Y_0}\,(<:_{\mathtt{ax}})}{s : Y'_0 <: Y_0 \vdash s^+ : Y'_0, T <: Y_0, T}\,(<:_+)}{\delta_1 : (Y_0, T) \rhd_\tau Y_1; s : Y'_0 <: Y_0 \vdash (\uparrow^{s^+} \delta_1) : Y'_0, T \rhd_\tau Y_1}$$

## 4. Catchable contexts

- **Case** $[\![F]\!]_E$. This case is similar to the case $[\![v]\!]_V$.

- **Case** $[\![\tilde{\mu}[x_i].\langle x_i \parallel F\rangle \tau']\!]_E$. In the source language, we have:

$$\dfrac{\Gamma, x_i : T, \Gamma' \vdash_F F : T^{\perp\!\!\!\perp} \quad \Gamma, x_i : T \vdash_\tau \tau' : \Gamma' \quad |\Gamma| = i}{\Gamma \vdash_E \tilde{\mu}[x_i].\langle x_i \parallel F\rangle \tau' : T^{\perp\!\!\!\perp}}$$

We have by induction hypothesis a proof of $\vdash [\![\tau']\!]_\tau : [\![\Gamma, x_i : T]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma$ and a proof $\Pi_F$ of $\vdash [\![F]\!]_F : [\![\Gamma, x_i : T, \Gamma']\!]_\Gamma \rhd_F T$. We can thus derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y \rhd_V T \vdash V : Y \rhd_t T}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash \mathtt{shift}^{|\tau'|+1}_{|\tau|} : (Y, T, [\![\Gamma']\!]_\Gamma) <: Y}}{V : Y \rhd_V T \vdash V\ \mathtt{shift}^{|\tau'|+1}_{|\tau|} : (Y, T, [\![\Gamma']\!]_\Gamma) \to (Y, T, [\![\Gamma']\!]_\Gamma) \rhd_F T \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi}_\tau}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y, V : Y \rhd_V T \vdash V\ \mathtt{shift}^{|\tau'|+1}_{|\tau|}\ \tau[\uparrow^t V]_t(\uparrow^{\sigma^+} [\![\tau']\!]_\tau) : (Y, T, [\![\Gamma']\!]_\Gamma) \rhd_F T \to \bot}\,(@) \quad \boldsymbol{\Pi}_F}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y, V : Y \rhd_V T \vdash V\ \mathtt{shift}^{|\tau'|+1}_{|\tau|}\ \tau[\uparrow^t V]_t(\uparrow^{\sigma^+} [\![\tau']\!]_\tau)\ (\uparrow^{\sigma'} [\![F]\!]_F) : \bot}\,(@)}{\vdash \lambda\sigma\tau V.V\ \mathtt{shift}^{|\tau'|+1}_{|\tau|}\ \tau[\uparrow^t V]_t(\uparrow^{\sigma^+} [\![\tau']\!]_\tau)\ (\uparrow^{\sigma'} [\![F]\!]_F) : [\![\Gamma]\!]_\Gamma \rhd_F T}\,(\lambda)$$

where:

– $k = |\tau'| + 1,\ \sigma' = \sigma^{k+}$
– $\Pi_F$ is the following proof, obtained by Corollary 15:

$$\dfrac{\vdash F : ([\![\Gamma]\!]_\Gamma, T, [\![\Gamma']\!]_\Gamma) \rhd_F T \quad \dfrac{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma^{k+} : Y, T, [\![\Gamma']\!]_\Gamma <: [\![\Gamma]\!]_\Gamma, T, [\![\Gamma']\!]_\Gamma}{\text{Lemma 7}}}{\sigma : Y <: [\![\Gamma']\!]_\Gamma \vdash (\uparrow^{\sigma'} F) : (Y, T, [\![\Gamma']\!]_\Gamma) \rhd_F T}$$

– $\Pi_\tau$ is the following proof:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{V : Y \rhd_V T \vdash V : Y \rhd_V T}\,(\mathrm{Ax})}{V : Y \rhd_V T \vdash \uparrow^t V : Y \rhd_t T}\,(\uparrow)}{\tau : Y, V : Y \rhd_V T \vdash \tau[\uparrow^t V]_t : Y, T}\,(\tau[t]) \qquad \boldsymbol{\Pi}_{\tau'}}{\tau : Y, V : Y' \rhd_V T; \sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \tau[\uparrow^t V]_t[\![\tau']\!]_\tau : (Y, T, [\![\Gamma']\!]^{\sigma[x:=n]})}\,(\tau\tau')$$

– $\Pi_{\tau'}$ is the following proof, obtained from the induction hypothesis for $\tau'$ and Corollary 15:

$$\dfrac{\vdash [\![\tau']\!]_\tau : [\![\Gamma]\!]_\Gamma, T \rhd_\tau [\![\Gamma']\!]_\Gamma \qquad \dfrac{\dfrac{}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma : Y <: [\![\Gamma]\!]_\Gamma}\,(<:\mathtt{ax})}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma^+ : Y, T <: [\![\Gamma]\!]_\Gamma, T}\,(<:_+)}{; \sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \uparrow^{\sigma^+} [\![\tau']\!]_\tau : Y, T \rhd_\tau [\![\Gamma']\!]_\Gamma}$$

## 5. Terms

- **Case** $[\![V]\!]_t$. This case is similar to the case $[\![v]\!]_V$.
- **Case** $[\![\mu\alpha_i.c]\!]_t$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\dfrac{\Gamma, \alpha_i : T^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = i}{\Gamma \vdash_t \mu\alpha_i.c : T}$$

Hence we have by induction a proof of $; \vdash [\![c]\!]_c : [\![\Gamma, x_i : T^{\perp\!\!\!\perp}]\!]_\Gamma \rhd_c \bot$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\vdash [\![c]\!]_c : [\![\Gamma, x_i : T^{\perp\!\!\!\perp}]\!]_\Gamma \rhd_c \bot \qquad \boldsymbol{\Pi}_\sigma}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y \vdash [\![c]\!]_c\, \sigma^+ : (Y, T^{\perp\!\!\!\perp}) \to \bot}\,(\forall_E) \qquad \boldsymbol{\Pi}_\tau}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y, E : Y \rhd_E T \vdash [\![c]\!]_c\, \sigma^+\, \tau[E]_E : \bot}\,(@)}{\vdash \lambda\sigma\tau E.[\![c]\!]_c\, \sigma^+\, \tau[E]_E : [\![\Gamma]\!]_\Gamma \rhd_t T}\,(\lambda)$$

where

– $\Pi_\sigma$ is the following derivation (since $|\tau|$ matches $|Y|$):

$$\dfrac{\dfrac{}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma : Y <: [\![\Gamma]\!]_\Gamma}\,(<:\mathtt{ax})}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma^+ : (Y, T^{\perp\!\!\!\perp}) <: [\![\Gamma, x_i : T^{\perp\!\!\!\perp}]\!]_\Gamma}\,(<:_+)$$

– $\Pi_E$ is also obtained by Lemma 14:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : \rhd_\tau Y}\,(\mathrm{Ax}) \quad \dfrac{}{E : Y \rhd_E T \vdash E : Y \rhd_E T}\,(\mathrm{Ax})}{\tau : Y, E : Y \rhd_E T; \vdash \tau[E]_E : \rhd_\tau Y, T^{\perp\!\!\!\perp}}\,(\tau[E])$$

## 6. Contexts

- **Case** $[\![E]\!]_e$. This case is similar to the case $[\![v]\!]_V$.
- **Case** $[\![\tilde{\mu} x_i.c]\!]_e$. This case is similar to the case $[\![\mu\alpha_i.c]\!]_t$.

**7. Commands**

- **Case** $[\![\langle t \parallel e\rangle]\!]_c$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus we have:

$$\frac{\Gamma \vdash_t t : T \quad \Gamma \vdash_e e : T^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \parallel e\rangle}$$

thus we get by induction two proofs of $; \vdash [\![t]\!]_t : [\![\Gamma]\!]_\Gamma \rhd_t T$ and $; \vdash [\![e]\!]_c : [\![\Gamma]\!]_\Gamma \rhd_e T$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\vdash [\![e]\!]_e : [\![\Gamma]\!]_\Gamma \rhd_e T \quad \overline{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma : Y <: [\![\Gamma]\!]_\Gamma}\ (<:_{\mathtt{ax}})}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y \vdash [\![e]\!]_e\ \sigma : Y \to Y \rhd_t T \to \bot}\ (\forall_E) \quad \dfrac{}{\tau : Y \vdash \tau : Y}\ (\mathrm{Ax})}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y \vdash [\![e]\!]_e\ \sigma\ \tau : Y \rhd_t T \to \bot}\ (\tau_E) \quad \boldsymbol{\Pi}_t}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y \vdash [\![e]\!]_e\ \sigma\ \tau\ (\uparrow^\sigma [\![t]\!]_t) : \bot}\ (@)}{\vdash \lambda\sigma\tau.[\![e]\!]_e\ \sigma\ \tau\ (\uparrow^\sigma [\![t]\!]_t) : [\![\Gamma]\!]_\Gamma \rhd_c \bot}\ (\lambda)$$

where:

– $\Pi_t$ is obtained using Lemma 8:

$$\frac{\vdash [\![t]\!]_t : [\![\Gamma]\!]_\Gamma \rhd_t T \quad \overline{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma : Y <: [\![\Gamma]\!]_\Gamma}\ (<:_{\mathtt{ax}})}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \uparrow^\sigma [\![t]\!]_t : Y \rhd_t T}$$

**8. Closures**

- **Case** $[\![c\tau']\!]_l$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\frac{\Gamma, \Gamma' \vdash_c c \quad \Gamma \vdash_\tau \tau' : \Gamma'}{\Gamma \vdash_l c\tau'}$$

where $n$ matches $|\Gamma|$. We thus get by induction two proofs $\vdash [\![\tau']\!]_\tau : [\![\Gamma]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma$ and $\vdash [\![c]\!]_c : [\![\Gamma, \Gamma']\!]_\Gamma \rhd_c \bot$. We can derive:

$$\dfrac{\dfrac{\vdash [\![c]\!]_c : [\![\Gamma, \Gamma']\!]_\Gamma \rhd_c \bot \quad \Pi_\sigma}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash [\![c]\!]_c\ \sigma' : (Y, [\![\Gamma']\!]_\Gamma) \to \bot}\ (\forall_E) \quad \dfrac{\overline{\tau : Y \vdash \tau : Y}\ (\mathrm{Ax}) \quad \boldsymbol{\Pi}'_\tau}{\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \tau(\uparrow^\sigma [\![\tau']\!]_\tau) : Y [\![\Gamma']\!]_\Gamma}\ (\tau\tau')}{\sigma : Y <: [\![\Gamma]\!]_\Gamma, \tau : Y \vdash [\![c]\!]_c\ \sigma'\ \tau(\uparrow^{\sigma'} [\![\tau']\!]_\tau) : \bot}\ (\tau_E)}{\vdash \lambda\sigma\tau.[\![c]\!]_c\ \sigma'\ \tau(\uparrow^\sigma [\![\tau']\!]_\tau)}\ (\lambda)$$

where $k = |\tau'|$, $\sigma' = \sigma^{k+}$ and:

– $\Pi_\sigma$ is a proof of $\sigma : Y <: [\![\Gamma]\!]_\Gamma \vdash \sigma^{k+} : (Y, [\![\Gamma']\!]_\Gamma) <: [\![\Gamma, \Gamma']\!]_\Gamma$ obtained by Lemma 7;
– $\Pi_{\tau'}$ is the following proof also obtained by Corollary 15:

$$\frac{\vdash [\![\tau']\!]_\tau : [\![\Gamma]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma \quad \overline{\vdash \sigma : Y <: [\![\Gamma]\!]_\Gamma}\ (<:_{\mathtt{ax}})}{\vdash (\uparrow^\sigma [\![\tau']\!]_\tau) : Y \rhd_\tau [\![\Gamma']\!]_\Gamma}$$

**9. Stores**

• **Case** $[\![\tau[x_i := t]]\!]_\tau$. We only consider the case $\tau[x_i := t]$, the proof for the case $\tau[\alpha_i := E]$ is identical. This corresponds to the typing rules:

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_t t : T \quad |\Gamma, \Gamma'| = i}{\Gamma \vdash_\tau \tau[x_i := t] : \Gamma', x_i : T}$$

By induction, we obtain two proofs of $\vdash [\![\tau]\!]_\tau : [\![\Gamma]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma$ and $\vdash [\![t]\!]_t : [\![\Gamma, \Gamma']\!]_\Gamma \rhd_t T$. We can thus derive by Lemma 14:

$$\frac{\vdash [\![\tau]\!]_\tau : [\![\Gamma]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma \quad \vdash [\![t]\!]_t : [\![\Gamma, \Gamma']\!]_\Gamma \rhd_t T}{\vdash [\![\tau]\!]_\tau[[\![t]\!]_t] : [\![\Gamma]\!]_\Gamma \rhd_\tau [\![\Gamma']\!]_\Gamma, T}\ (\tau[t])$$

$$\llbracket \lambda x_i.t \rrbracket_V\ \sigma\ \tau\ u\ E \triangleq \llbracket t \rrbracket_t\ \sigma^+\ \tau[u]\ \uparrow^{\mathtt{shift}^1_{|\tau|}} E \qquad \llbracket \boldsymbol{k} \rrbracket_V \triangleq \boldsymbol{k}$$

$$\llbracket t \cdot E \rrbracket_E\ \sigma\ \tau\ v \triangleq v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t)\ (\uparrow^\sigma \llbracket E \rrbracket_E) \qquad \llbracket \boldsymbol{\kappa} \rrbracket_E \triangleq \boldsymbol{\kappa}$$

$$\llbracket \alpha_i \rrbracket_E\ \sigma\ \tau\ V \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\ x\ (\mathtt{shift}^{|\delta_1|+1}_{|\delta_0|})\ \tau\ V$$

$$\llbracket V \rrbracket_t\ \sigma\ \tau\ E \triangleq E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket V \rrbracket_V) \qquad \llbracket \mu\alpha_i.c \rrbracket_t\ \sigma\ \tau\ E \triangleq \llbracket c \rrbracket_c\ \sigma^+\ \tau[E]_E$$

$$\llbracket x_i \rrbracket_t\ \sigma\ \tau\ E \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\ x\ (\mathtt{shift}^{|\delta_1|+1}_{|\delta_0|})\ \tau\ E$$

$$\llbracket E \rrbracket_e\ \sigma\ \tau\ t \triangleq t\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket E \rrbracket_E) \qquad \llbracket \tilde{\mu} x_i.c \rrbracket_e\ \sigma\ \tau\ t \triangleq \llbracket c \rrbracket_c\ \sigma^+\ \tau[t]_t$$

$$\llbracket \langle t \parallel e \rangle \rrbracket_c\ \sigma\ \tau \triangleq \llbracket e \rrbracket_e\ \sigma\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t)$$

$$\llbracket c\tau \rrbracket^n_l\ \sigma\ \tau' \triangleq \llbracket c \rrbracket_c\ \sigma'\ \tau'(\uparrow^{\sigma'} \llbracket \tau \rrbracket_\tau) \quad \text{where } k = |\tau'| - n,\ \sigma' = \sigma^{k+}$$

$$\llbracket \tau_0[x_i := t] \rrbracket_\tau \triangleq \llbracket \tau_0 \rrbracket_\tau[\llbracket t \rrbracket_t]_t \qquad \llbracket \tau_0[\alpha_i := E] \rrbracket_\tau \triangleq \llbracket \tau_0 \rrbracket_\tau[\llbracket E \rrbracket_E]_E$$

$$\llbracket \varepsilon \rrbracket_\tau \triangleq \varepsilon$$

(a) Translation of terms

$$\llbracket \Gamma \vdash_e e : T^{\perp\!\!\!\perp} \rrbracket \triangleq \vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket_\Gamma \rhd_e T \qquad \llbracket \Gamma \vdash_c c \rrbracket \triangleq \vdash \llbracket c \rrbracket_c : \llbracket \Gamma \rrbracket_\Gamma \rhd_c \bot$$

$$\llbracket \Gamma \vdash_t t : T \rrbracket \triangleq \vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \rhd_t T \qquad \llbracket \Gamma \vdash_l l \rrbracket \triangleq \vdash \llbracket l \rrbracket^{|\Gamma|}_l : \llbracket \Gamma \rrbracket_\Gamma \rhd_c \bot$$

$$\llbracket \Gamma \vdash_E E : T^{\perp\!\!\!\perp} \rrbracket \triangleq \vdash \llbracket E \rrbracket_E : \llbracket \Gamma \rrbracket_\Gamma \rhd_E T \qquad \llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket \triangleq \vdash \llbracket \tau \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \rhd_\tau \llbracket \Gamma' \rrbracket_\Gamma$$

$$\llbracket \Gamma \vdash_V V : T \rrbracket \triangleq \vdash \llbracket V \rrbracket_V : \llbracket \Gamma \rrbracket_\Gamma \rhd_V T$$

$$\Upsilon \rhd_c T \triangleq \forall Y <: \Upsilon.\, Y \to \bot \qquad \Upsilon \rhd_V X \triangleq X$$

$$\Upsilon \rhd_e T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_t T) \to \bot \qquad \Upsilon \rhd_V T \to U \triangleq$$

$$\Upsilon \rhd_t T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_E T) \to \bot \qquad \forall Y <: \Upsilon.\, Y \to (Y \rhd_t T) \to (Y \rhd_E U) \to \bot$$

$$\Upsilon \rhd_E T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_V T) \to \bot$$

(b) Translation of types and judgments

**Fig. 16.** Call-by-name continuation-and-environment-passing style translation

## F A typed call-by-name translation

We first rephrase the reduction rules to use De Bruijn levels:

$$\begin{array}{rcl}
\langle t \parallel \tilde{\mu} x_i.c \rangle \tau & \to & c[x_n/x_i]\tau[x_n := t] \text{ with } |\tau| = n \\
\langle \mu\alpha_i.c \parallel E \rangle \tau & \to & c[\alpha_n/\alpha_i]\tau[\alpha_n := E] \text{ with } |\tau| = n \\
\langle x_n \parallel E \rangle \tau & \to & \langle \tau(n) \parallel E \rangle \tau \\
\langle V \parallel \alpha_n \rangle \tau & \to & \langle V \parallel \tau(n) \rangle \tau \\
\langle \lambda x_i.t \parallel u \cdot E \rangle \tau & \to & \big\langle u \,\big\|\, \tilde{\mu} x_i.\langle t \parallel E \rangle \big\rangle \tau
\end{array}$$

We give in Fig. 16 the full translation for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environment. Once again, we have:

**Theorem 11.** The translation is well-typed, *i.e.*:

1. If $\Gamma \vdash_V V : T$ then $[\![\Gamma \vdash_V V : T]\!]$
2. If $\Gamma \vdash_E E : T^{\perp\!\!\!\perp}$ then $[\![\Gamma \vdash_E E : T^{\perp\!\!\!\perp}]\!]$
3. If $\Gamma \vdash_t t : T$ then $[\![\Gamma \vdash_t t : T]\!]$
4. If $\Gamma \vdash_e e : T^{\perp\!\!\!\perp}$ then $[\![\Gamma \vdash_e e : T^{\perp\!\!\!\perp}]\!]$
5. If $\Gamma \vdash_c c$ then $[\![\Gamma \vdash_c c]\!]$
6. If $\Gamma \vdash_l l$ then $[\![\Gamma \vdash_l l]\!]$
7. If $\Gamma \vdash_\tau \tau$ then $[\![\Gamma \vdash_\tau \tau : \Gamma']\!]$

*Proof.* The proof is very similar (and easier) than the proof in the call-by-need case, by induction on typing derivations. In particular, all the lemmas proved in Section 4.1 also hold for the call-by-name translation.

It is interesting to observe that even though terms are stored once and for all in call-by-name, the use of a global environment forces us to quantify over arbitrary extensions of the store. Indeed, through the translation each (typed) term $t$ is waiting for a store whose type should match its former typing context. Yet, many computations may happen before $[\![t]\!]_t$ is evaluated, corresponding to other branches of the global typing derivation. As a consequence, the store may contain arbitrarily more elements at that time.

*Example 5.* Consider a term $x : A, y : B \vdash_t u : C$, through the translation we will thus have $\vdash [\![u]\!]_t : A, B \rhd_t C \to D$. Now, imagine that we dispose of three values $V_0, V_1, V_2$ respectively of types $A, B, C$, we can thus construct three closed terms $t_0, t_1, t_2$ such that, given a continuation, $t_i$ is going to produce arbitrary computations (and in particular store arbitrarily many terms, let us denote the resulting store by $\tau_i : U_i$) before returning $V_i$ to its continuation. These terms can thus be assigned the types $(A \to D) \to D$, $(B \to D) \to D$, $(C \to D) \to D$, and the closed term $t_u \triangleq t_0(\lambda x.t_1(\lambda y.t_2 u))$ can thus be typed by $\vdash t_u : D$. Now, if $t_u$ is evaluated in an initially empty store, at the moment where $uV_2$ will be evaluated, the store will be $\tau_0[x := V_0]\tau_1[y := V_1]\tau_2$ of type $U_0, A, U_1, B, U_2$.

## G A typed call-by-value translation

To illustrate the generality of our construction, we give one more example by giving a typed continuation-and-environment-passing style translation from the call-by-value $\bar{\lambda}\mu\tilde{\mu}$-calculus with explicit environments. The syntax of the calculus is given by:

$$\begin{array}{llll} \textbf{Values} & V ::= \lambda x_i.t \mid x_i \mid \boldsymbol{k} & \textbf{Co-values} & E ::= t \cdot e \mid \alpha_i \mid \boldsymbol{\kappa} \\ \textbf{Terms} & t, u ::= V \mid \mu\alpha_i.c & \textbf{Contexts} & e ::= E \mid \tilde{\mu}x_i.c \end{array}$$

$$\begin{array}{ll} \textbf{Environment} & \tau ::= \varepsilon \mid \tau[x_i := V] \mid \tau[\alpha_i := E] \\ \textbf{Commands} & c ::= \langle t \parallel e \rangle \\ \textbf{Closures} & l ::= c\tau \end{array}$$

while the reduction rules are given by:

$$\begin{array}{llll} (\textsc{Catch}) & \langle \mu\alpha_i.c \parallel e \rangle\tau & \to & c[\alpha_n/\alpha_i]\tau[\alpha_n := e] \text{ with } |\tau| = n \\ (\textsc{Let}) & \langle V \parallel \tilde{\mu}x_i.c \rangle\tau & \to & c[x_n/x_i]\tau[x_n := V] \text{ with } |\tau| = n \\ (\textsc{Lookup}_x) & \langle V \parallel \alpha_n \rangle\tau & \to & \langle V \parallel \tau(n) \rangle\tau \\ (\textsc{Lookup}_\alpha) & \langle x_n \parallel E \rangle\tau & \to & \langle \tau(n) \parallel E \rangle\tau \\ (\textsc{Beta}) & \langle \lambda x_i.t \parallel u \cdot e \rangle\tau & \to & \langle u \parallel \tilde{\mu}x_i.\langle t \parallel e \rangle \rangle\tau \end{array}$$

Since only values can be stored in environments, we will use the following parameters for $F_\Upsilon$:

$$\frac{\Gamma \vdash t : \Upsilon \rhd_V T}{\Gamma \vdash [t]_t : \Upsilon \rhd_\tau T}\,(\tau_t) \qquad \frac{\Gamma \vdash t : \Upsilon \rhd_E T}{\Gamma \vdash [t]_E : \Upsilon \rhd_\tau T^{\perp\!\!\!\perp}}\,(\tau_E)$$

The translation of terms, which is naturally obtained by first examinating a small steps reduction system, is defined by:

$$\begin{array}{rcl} [\![\lambda x_i.t]\!]_V\ \sigma\ \tau\ u\ E & \triangleq & u\ \mathtt{id}_{|\tau|}\ \tau\ (\lambda s\delta v.[\![t]\!]_t\ (s \circ \sigma)^+\ \tau[v]\ \uparrow^s E \\ [\![x_i]\!]_V\ \sigma\ \tau & \triangleq & \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\ x\ (\mathtt{shift}_{|\delta_0|}^{|\delta_1|+1})\ \tau \\ [\![\boldsymbol{k}]\!]_V & \triangleq & \boldsymbol{k} \end{array}$$

$$\begin{array}{rcl} [\![t \cdot e]\!]_e\ \sigma\ \tau\ V & \triangleq & V\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![u]\!]_t)\ (\uparrow^\sigma [\![e]\!]_e) \\ [\![\alpha_i]\!]_E\ \sigma\ \tau\ V & \triangleq & \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ \delta_0, x, \delta_1\ \mathtt{in}\ x\ (\mathtt{shift}_{|\delta_0|}^{|\delta_1|+1})\ \tau\ V \\ [\![\tilde{\mu}x_i.c]\!]_e\ \sigma\ \tau\ t & \triangleq & [\![c]\!]_c\ \sigma^+\ \tau[t]_t \\ [\![\boldsymbol{\kappa}]\!]_E & \triangleq & \boldsymbol{\kappa} \end{array}$$

$$\begin{array}{rcl} [\![V]\!]_t\ \sigma\ \tau\ e & \triangleq & e\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma [\![V]\!]_V) \\ [\![\mu\alpha_i.c]\!]_t\ \sigma\ \tau\ E & \triangleq & [\![c]\!]_c\ \sigma^+\ \tau[E]_E \end{array}$$

$$\begin{array}{rcl} [\![\langle t \parallel e \rangle]\!]_c\ \sigma\ \tau & \triangleq & [\![e]\!]_e\ \sigma\ \tau\ (\uparrow^\sigma [\![t]\!]_t) \\ [\![c\tau]\!]_l^n\ \sigma\ \tau' & \triangleq & [\![c]\!]_c\ \sigma'\ \tau'(\uparrow^{\sigma'} [\![\tau]\!]_\tau) \\ & & \text{where } k = |\tau'| - n,\ \sigma' = \sigma^{k+} \end{array}$$

$$\begin{array}{rcl} [\![\tau_0[x_i := t]]\!]_\tau & \triangleq & [\![\tau_0]\!]_\tau[[\![t]\!]_t]_t \\ [\![\tau_0[\alpha_i := E]]\!]_\tau & \triangleq & [\![\tau_0]\!]_\tau[[\![E]\!]_E]_E \\ [\![\varepsilon]\!]_\tau & \triangleq & \varepsilon \end{array}$$

As for the translations of types and judgments, it is very similar to the translation in the call-by-name and call-by-need settings, but adapted to match the alternation of levels in the translation of terms. Namely, since terms at level $t$ are first analyzed, then context at level $e$, then values, the translation follows the same hierarchy:

$$\begin{array}{l|l}
\llbracket \Gamma \vdash_t t : T \rrbracket \triangleq \vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \rhd_t T & \llbracket \Gamma \vdash_c c \rrbracket \triangleq \vdash \llbracket c \rrbracket_c : \llbracket \Gamma \rrbracket_\Gamma \rhd_c \bot \\
\llbracket \Gamma \vdash_e e : T^{\perp\!\!\!\perp} \rrbracket \triangleq \vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket_\Gamma \rhd_e T & \llbracket \Gamma \vdash_l l \rrbracket \triangleq \vdash \llbracket l \rrbracket_l^{|\Gamma|} : \llbracket \Gamma \rrbracket_\Gamma \rhd_c \bot \\
\llbracket \Gamma \vdash_V V : T \rrbracket \triangleq \vdash \llbracket V \rrbracket_V : \llbracket \Gamma \rrbracket_\Gamma \rhd_V T & \llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket \triangleq \vdash \llbracket \tau \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \rhd_\tau \llbracket \Gamma' \rrbracket_\Gamma \\
\llbracket \Gamma \vdash_E E : T^{\perp\!\!\!\perp} \rrbracket \triangleq \vdash \llbracket E \rrbracket_E : \llbracket \Gamma \rrbracket_\Gamma \rhd_E T &
\end{array}$$

---

$$\begin{array}{l|l}
\Upsilon \rhd_c T \triangleq \forall Y <: \Upsilon. Y \to \bot & \Upsilon \rhd_E X \triangleq X \\
\Upsilon \rhd_t T \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_e T) \to \bot & \Upsilon \rhd_E T \to U \triangleq \\
\Upsilon \rhd_e T \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_V T) \to \bot & \forall Y <: \Upsilon. Y \to (Y \rhd_V T) \to (Y \rhd_t U) \to \bot \\
\Upsilon \rhd_V T \triangleq \forall Y <: \Upsilon. Y \to (Y \rhd_E T) \to \bot &
\end{array}$$

Once again, we can check that:

**Theorem 17.** *The translation is well-typed, i.e.:*

1. *If* $\Gamma \vdash_V V : T$ *then* $\llbracket \Gamma \vdash_V V : T \rrbracket$
2. *If* $\Gamma \vdash_e e : T^{\perp\!\!\!\perp}$ *then* $\llbracket \Gamma \vdash_e e : T^{\perp\!\!\!\perp} \rrbracket$
3. *If* $\Gamma \vdash_t t : T$ *then* $\llbracket \Gamma \vdash_t t : T \rrbracket$
4. *If* $\Gamma \vdash_c c$ *then* $\llbracket \Gamma \vdash_c c \rrbracket$
5. *If* $\Gamma \vdash_l l$ *then* $\llbracket \Gamma \vdash_l l \rrbracket$
6. *If* $\Gamma \vdash_\tau \tau$ *then* $\llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket$