A calculus of expandable stores

Hugo Herbelin, Étienne Miquey

A calculus of expandable stores

Continuation-and-environment-passing style translations

Hugo Herbelin
INRIA, IRIF (CNRS)
Université Paris-Diderot
France
hugo.herbelin@inria.fr

Étienne Miquey
CNRS, LSV, INRIA
ÉNS Paris-Saclay
France
emiquey@lsv.fr

Abstract

The call-by-need evaluation strategy for the λ-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, facilitating 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 translation, that is to say how to soundly translate in continuation-passing style a calculus with global environment. To this end, we introduce F_{Υ}, a generic calculus to define the target of such translations. In particular, F_{Υ} features a data type for typed stores and a mechanism of explicit coercions witnessing store extensions along environment-passing style translations. On the logical side, this broadly amounts to a Kripke forcing-like translation mixed with a negative translation (for the continuation-passing part). Since F_{Υ} allows for the definition of such translations for different source calculi (call-by-need, call-by-name, call-by-value) with different type systems (simple types, system F), we claim that it precisely captures the computational content of continuation-and-environment-passing style translations.

Keywords global environment, abstract machines, CPS translations, Kripke forcing, lazy evaluation, de Bruijn indices

This paper is an extended version of a conference paper available at: https://dx.doi.org/10.1145/3373718.3394792. Due to the original space constraints, most of the proof are given in appendices.

Introduction

Variable binding is one of these notions that seems really natural on the surface (especially to humans that have some habit of working with variables), while being much more subtle to define formally (computer scientists are well aware of that fact). In particular, the way programming languages handle bindings and environments has evolved a lot since the emergence of computers.

In the realm of the λ-calculus, most of the standard abstract machines evaluate terms within a local environment by using closures (*e.g.* Landin's SECD machine [20], the Krivine Abstract Machine [18], *etc.*). Nonetheless, these machines do not allow the use of evaluation strategies such as call-by-need that require the ability to share computations. Instead, lazy evaluation strategies require abstract machines with a global environment [3, 33]. Such machines demand in particular to explicitly handle addresses (as in the Lazy Krivine [21]) or a renaming process (as in the Milner Abstract Machine [2]).

Continuation-passing style translations, that where first introduced by Sussman and Steel [35], constitute a great tool when it comes to studying operational semantics of calculi: by making explicit the order in which reduction steps are computed, CPS translations indirectly specify an evaluation strategy for the translated calculus. On a logical aspect, they are also very informative insofar as the translation they induce at the level of types mostly amounts to a syntactic model allowing to transfer logical properties (coherence, normalization) from the target calculus [6]. For instance, standard CPS translations are known to correspond to embeddings of classical logic into intuitionistic logic [14, 28].

If there exists a wide literature on (typed) CPS translations for call-by-name or call-by-value calculi [4, 13, 32], the situation is quite different for lazily evaluated calculi. Even though such strategies are used in practice, for instance in Haskell or Coq, no typed CPS allows to take them into account. The main difficulty in deriving a CPS translation for a call-by-need calculus is related to the necessity of a global environment: since the evaluation of terms is shared, the CPS translation actually has to be combined with an environment-passing style transformation. Also, since environments can grow during the execution, such a translation needs to handle this extensibility. Last but not least, the use of a global environment requires tackling problems related to the uniqueness of names.

This paper aims at bridging the gap between CPS translations and calculi with a global environment. Our main result is the introduction of F_{Υ}, a highly parametric calculus suitable as the target of several typed continuation-and-environment-passing style (CEPS) translations, taking into account source

calculi with different evaluation strategies or type systems. It features the architecture to manipulate typed expandable stores, whose elements are accessed via the careful use of de Bruijn levels. Store extensions are computationally witnessed by explicit coercions which allow us to update de Bruijn levels along the execution. In some sense, we make explicit the rewriting system underlying store management, in a similar way as explicit substitution calculi, as e.g. in [1], make explicit the computational structure of substitution.

Our results improve over the state-of-the-art in a previous paper by Ariola *et al.*, in which they define an untyped CPS translation for a call-by-need calculus with control [5]. Besides just being able to type this translation, our setting solves a subtle issue of renaming in their translation. The de Bruin presentation used in this paper formally addresses this issue by avoiding names altogether. Most importantly, this paper shows that a de Bruin version of [5] is even possible. This fact was not obvious from the previous work, which depends deeply on name-based binding and lookup operations on an environment. Insofar as name-based environments are neither very practical nor efficient for most applications, demonstrating how the system of [5] can be statically compiled to numeric offsets into an ordered environment is a step forward toward a practical implementation strategy for call-by-need with control effects.

Let us begin this paper with a comprehensive introduction to abstract machines with global environments. This introduction shall then lead us to analyzing the technicalities that CEPS translations give rise to (Sec. 2) before introducing F_{Υ} more in depth (Sec. 3). and illustrating its features by defining different typed CEPS (Sec. 4). We shall conclude with further perspectives arising from this work (Sec. 5).

1 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 the following examples, 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.

1.1 The Milner Abstract Machine

Even though it is far from being the most well-known abstract machine, the Milner Abstract Machine (MAM) is probably the easiest presentation of an abstract machine for the (call-by-name) λ-calculus that uses a single global environment [2, 3]. A state of this machine is made of three components:

- a code $\overline{t}$ for a term t which is not considered up to α-conversion[1],
- a stack π which contains the arguments of the current code, that is to say a stack of codes,
- a global environment τ, which is a list storing the (delayed) substitutions generated by the redexes encountered so far.

Formally, this corresponds to the following syntax:

Terms	$t, u ::= x \mid tu \mid \lambda x.t$
Stacks	$\pi ::= \varepsilon \mid \overline{t} \cdot \pi$
Environments	$\tau ::= \varepsilon \mid \tau[x := t]$

The machine is given in Figure 1, where we follow the notations of Accattoli and Barras [3] to denote reduction rules: we write $\to_\beta$ for the β-reduction rule, $\to_{\mathsf{s}}$ for the rule which somehow performs a *substitution*, and $\to_{\mathsf{c}}$ 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 λ-calculus [2].

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 α-renaming process. Executing twice a term binding 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 $\to_{\mathsf{s}}$ rule that if x is linked to some term t in the environment τ, accessing x results in executing a code $\overline{t}^{\alpha}$ that is α-equivalent to t and such that any bound name in $\overline{t}^{\alpha}$ is fresh with respect to every other names in the term, in the current stack π and in the environment τ. 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 λ-calculus use many local environments (*e.g.* the Krivine Abstract Machine [18], Landin's SECD machine [20], Felleisen and Friedman's CEK machine [12], Leroy's ZINC machine [22]). 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

[1] In other words, we assume implicit α-conversion for terms, which stand for the equivalence class of raw syntactic codes up to α-conversion.

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

(a) Milner Abstract Machine

$$
\begin{array}{lll}
\overline{t}\overline{u} \star S \star E & \to_{\mathsf{c}} & \overline{t} \star (\overline{u}, E)\cdot S \star E \\
\lambda x.\overline{t} \star (\overline{u}, E')\cdot S \star E & \to_{\beta} & \overline{t} \star S \star E[x ::= (\overline{u}, E')] \\
x \star S \star E[x ::= (\overline{t}, E')]E'' & \to_{\mathsf{s}} & \overline{t} \star S \star E'
\end{array}
$$

(b) Krivine Abstract Machine

Figure 1. MAM vs KAM

Abstract Machine (KAM), whose syntax is given by:

Terms	$t, u ::= x \mid tu \mid \lambda x.t$
Stacks	$S ::= \varepsilon \mid (t, E)\cdot S$
Environments	$E ::= \varepsilon \mid E[x ::= (t, E')]$

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 α-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 [3]), it has the drawback of being incompatible with lazy evaluation strategies which require to share computations and memory bindings.

1.2 The Milner Abstract Machine By-Need

The call-by-need evaluation strategy of the λ-calculus evaluates arguments of functions only when needed, and, when needed, shares their evaluations across all places where the argument is required. Therefore, abstract machines implementing the call-by-need evaluation have to allow for some kind of global environment in order to share computations [8, 33]. The Milner Abstract Machine can easily be modified to obtain such an abstract machine, called the *Milner Abstract machine by-neeD* (MAD) [2]. 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 τ_1 while saving the current stack together with the rest of the environment τ_2 and the variable x on the dump. Then, if this computation eventually produces a value v in an environment τ_1', the machine goes back to the former computation within the updated environment $\tau_1'[x := v]\tau_2$. The machine is given Figure 2, where environments τ and stacks π are defined as in the MAM, and where dumps are given by the following grammar:

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

Alternatively, one could also extend the syntax of stacks in order to add a constructor $\mathsf{h}(x, \tau)\cdot\pi$ that holds the entries

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

Figure 2. The Milner Abstract machine by-neeD

that would otherwise go to the dump. The modified reduction rules could then be given by:

$$
\begin{array}{lll}
x \star \pi \star \tau[x := \overline{t}]\tau' & \to_{\mathsf{c}_2} & \overline{t} \star \mathsf{h}(x, \tau')\cdot\pi \star \tau \\
\overline{v} \star \mathsf{h}(x, \tau')\cdot\pi \star \tau & \to_{\mathsf{s}} & \overline{v}^{\alpha} \star \pi \star \tau[x := \overline{v}]\tau'
\end{array}
$$

This is for instance the approach chosen in Sestoft's machine for lazy evaluation [33] or in Accattoli, Barenbaum and Mazza's *Merged MAD* [2]. For a more detailed introduction to the subtleties related to the implementation of lazy abstract machines or a comparison between the different approaches (including Crégut's lazy KAM [8] and Sestoft's abstract machine), we refer the reader to Accattoli, Barenbaum and Mazza's paper [2].

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

Let us briefly explain how the MAM could be expressed under the shape of a sequent calculus. We shall dwell on a by-need variant of this calculus in Section 1.4.

We present here a variant of the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus extended with a global environment [9]. 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 $\tilde{\mu}x.c$ can be read as a context $\mathtt{let}\, x = [\,]\, \mathtt{in}\, c$. As for terms, the μ operator comes from Parigot's $\lambda\mu$-calculus [30]: $\mu\alpha$ binds a context to a context variable α in the same way $\tilde{\mu}x$ binds a proof to some proof variable x. In particular, as we shall see now, it allows terms to capture evaluation contexts and as such plays the role of a control operator.

The syntax and reduction rules are given in Figure 3, in which terms and contexts are implicitly considered up to α-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[2]. As per [9], we restrict the

[2]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 λ-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.

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

(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)	$\langle \lambda x.t \parallel u \cdot E\rangle\tau$	$\to$	$\left\langle u \,\middle\|\, \tilde{\mu}x.\langle t \parallel E\rangle\right\rangle\tau$

Figure 3. By-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environments

syntax of stacks to $t \cdot E$ for call-by-name evaluation rather than the more general te. Observe that the reduction rules for μ is restricted to co-values, which enforces the call-by-name reduction strategy. Call-by-value is obtained by relaxing these constraints and by dually constraining the reduction of $\tilde{\mu}$ to values [9]. Note in particular our formulation of the (Beta) rule: by delegating the process of substitution of the argument to the $\tilde{\mu}$ rule, it captures what is common to call-by-name β and call-by-value β_V, leaving the rôle of choosing between call-by-name and call-by-value to the rules μ and $\tilde{\mu}$. 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 the next section, we thus let them as an exercise for the reader.

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

Although 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 λ-calculus turns out to be trickier than the call-by-value and call-by-name cases. In particular, a first attempt by Okasaki, Lee, Tarditi [29] was latter shown to be non-normalizing on simply-typed terms [5].

In the latter, Ariola *et al.* 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[3]. 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 1.3). 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*).

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 detail, 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 $\tilde{\mu}[x].\langle x \parallel F\rangle\tau'$:

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

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 τ 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 τ' that was originally located after the binding $[x := \ldots]$. 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:

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

The brackets in $\tilde{\mu}[x].c$ are used to express the fact that the variable x is forced at top-level[4]. 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[5].

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

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

[5] 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.

Strong values	$v ::= \boldsymbol{k} \mid \lambda x_i.t$
Weak values	$V ::= v \mid x_i$
Terms	$t, u ::= V \mid \mu\alpha_i.c$
Stores	$\tau ::= \varepsilon \mid \tau[x_i := t] \mid \tau[\alpha_i := E]$
Commands	$c ::= \langle t \parallel e \rangle$
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)	$\langle t \parallel \tilde{\mu}x_i.c\rangle\tau$	$\to^\dagger$	$c[x_n/x_i]\tau[x_n := t]$
(Catch)	$\langle \mu\alpha_i.c \parallel E\rangle\tau$	$\to^\dagger$	$c[\alpha_n/\alpha_i]\tau[\alpha_n := E]$
(Lookup$_\alpha$)	$\langle V \parallel \alpha_n\rangle\tau$	$\to$	$\langle V \parallel \tau(n)\rangle\tau$
(Lookup$_x$)	$\langle x_n \parallel F\rangle\tau_0[x_n := t]\tau_1$	$\to$	$\big\langle t \,\big\|\, \tilde{\mu}[x_n].\langle x_n \parallel F\rangle\tau_1\big\rangle\tau_0$
(Restore)	$\big\langle V \,\big\|\, \tilde{\mu}[x_i].\langle x_i \parallel F\rangle\tau'\big\rangle\tau$	$\to^\dagger$	$\langle V \parallel \overline{F}\rangle\tau[x_n := V]\overline{\tau'}^\ddagger$
(Beta)	$\langle \lambda x_i.t \parallel u \cdot E\rangle\tau$	$\to^\dagger$	$\big\langle u \,\big\|\, \tilde{\mu}x_n.\langle t[x_n/x_i] \parallel E\rangle\big\rangle\tau$

† with $|\tau| = n$ $\quad$ ‡ $\overline{F} \triangleq \uparrow_i^{+(n-i)} F$ and $\overline{\tau'} \triangleq \uparrow_i^{+(n-i)} \tau'$

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

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.

1.5 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 α-conversion on-the-fly. While this assumption, which is crucial to ensure the freshness of bindings added to the environment, is straightforward in the abstract machine model, it is not possible in the CPS translation [6]. In turn, we will use de Bruijn levels[7] 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 notation x_i to denote a term variable of de Bruijn level i where x is a fixed name whose sole purpose is to remind that the variable is binding terms. For binders of evaluation contexts, we similarly use de Bruijn levels, but with variables of the form α_i, where, again, α is a fixed name whose sole purpose is to remind that the variable is binding evaluation contexts, and the relevant information is the index i. The corresponding syntax is given in Figure 4. Note that typing rules for binders include a possible weakening of the context. In particular, giving a meaning to de Bruijn levels requires the knowledge of the length of the current context.

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 $\uparrow_i^{+n} t$ as the term t where the free variables x_j (resp. α_j) with $j \geq i$ have been replaced by x_{j+n} (resp. α_{j+n})[8]. The reduction rules are given in Figure 4. Note that we choose to perform index 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 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 $v, V, t, F, E, e, l, c, \tau$:

$\Gamma \vdash_l l$	$\Gamma \vdash_t t : T$	$\Gamma \vdash_e e : T^\perp$
$\Gamma \vdash_c c$	$\Gamma \vdash_V V : T$	$\Gamma \vdash_E E : T^\perp$
$\Gamma \vdash_\tau \tau : \Gamma'$	$\Gamma \vdash_v v : T$	$\Gamma \vdash_F F : T^\perp$

where types and typing contexts are defined by:

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

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 $T^\perp$; 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 [26], 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 1.1 (Subject reduction). *If* $\Gamma \vdash_l c\tau$ *and* $c\tau \to c'\tau'$ *then* $\Gamma \vdash_l c'\tau'$.

[6] See Appendix B.2 for more details on problems related to α-conversion

[7] De Bruijn levels were originally introduced as a reversed notation for de Bruijn indices [11]. While they are less common in the literature, levels have the significant benefit that variables referring to the same binder have the same name.

[8] See Appendix B.3 for a formal definition.

$$\frac{(\boldsymbol{k} : X) \in \mathcal{S}}{\Gamma \vdash_v \boldsymbol{k} : X} \qquad \frac{\Gamma, x_n : T \vdash_t t : U \quad |\Gamma| = n}{\Gamma, \Gamma' \vdash_v \lambda x_n.t : T \to U} \qquad \frac{(\boldsymbol{\kappa} : X) \in \mathcal{S}}{\Gamma \vdash_F \boldsymbol{\kappa} : X^\perp} \qquad \frac{\Gamma \vdash_t t : T \quad \Gamma \vdash_E E : U^\perp}{\Gamma \vdash_F t \cdot E : (T \to U)^\perp} \qquad \frac{\Gamma(n) = (x_n : T)}{\Gamma \vdash_V x_n : T} \qquad \frac{\Gamma \vdash_v v : T}{\Gamma \vdash_V v : T}$$

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

$$\frac{\Gamma, \alpha_n : T^\perp \vdash_c c \quad |\Gamma| = n}{\Gamma, \Gamma' \vdash_t \mu\alpha_n.c : T} \qquad \frac{\Gamma \vdash_E E : T^\perp}{\Gamma \vdash_e E : T^\perp} \qquad \frac{\Gamma, x_n : T \vdash_c c \quad |\Gamma| = n}{\Gamma, \Gamma' \vdash_e \tilde{\mu} x_n.c : T^\perp} \qquad \frac{\Gamma \vdash_t t : T \quad \Gamma \vdash_e e : T^\perp}{\Gamma \vdash_c \langle t \parallel e\rangle} \qquad \frac{\Gamma, \Gamma' \vdash_c c \quad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}$$

$$\frac{}{\Gamma \vdash_\tau \varepsilon : \varepsilon} \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_t t : T \quad |\Gamma, \Gamma'| = n}{\Gamma \vdash_\tau \tau[x_n := t] : \Gamma', x_n : T} \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_E E : T^\perp \quad |\Gamma, \Gamma'| = n}{\Gamma \vdash_\tau \tau[\alpha_n := E] : \Gamma', \alpha_n : T^\perp}$$

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

Proof. The proof proceeds by induction on typing derivation, and is almost the same as in the case without de Bruijn levels [26, Theorem 1]. □

2 Towards typed CEPS translations

We shall now introduce System F_Υ, the calculus that we will use as the target of several continuation-and-environment-passing style translations. Our main objective here is to identify the core ingredients necessary to the definition of a generic target calculus, independently of the source languages we will consider afterwards. In particular, our calculus will be suitable to express translation of calculi with different evaluation strategies or different type systems. As we shall see, System F_Υ is essentially Cardelli's System F $_<:$ [7] extended with the necessary structure to handle typed (extensible) stores.

In order to ease the introduction of System F_Υ, we first focus on the case of the simply typed $\overline{\lambda}_{[lv\tau\star]}$-calculus. We begin this section by explaining the rationale guiding the translation of types (which we only outline in this section) while taking advantage of this overview to highlight the different elements that such a translation requires.

2.1 Guidelines of the translation

Let us start by introducing step by step the intuitions guiding the definition of the translation. The main idea is that, due to 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. As we observed in Section 1, the environment is extensible, therefore, to anticipate extensions 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[9] $a : A, \alpha : A^\perp, b : B \vdash_e e : C$.

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 emphasized in [5, 26], there are 6 different levels of control in the $\overline{\lambda}_{[lv\tau\star]}$-calculus (corresponding to the 6 nested syntactic categories), leading to 6 mutually defined levels of interpretation. We define $[\![A \to B]\!]_v$ for strong values as $[\![A]\!]_t \to [\![B]\!]_t$, 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 $[\![A]\!]_e =^{5}_{\neg} [\![A]\!]_v$ (where we use the notations $^{1}_{\neg} A \triangleq A \to \bot$ and $^{n+1}_{\neg} A \triangleq \neg\, ^{n}_{\neg} A$).

As observed in the realizability interpretation [26], hypotheses from a context Γ of the form $\alpha : A^\perp$ 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 Γ by $[\![\Gamma]\!]$, in the particular case of $\Gamma \vdash_t A$ the translation is $[\![\Gamma]\!] \vdash [\![A]\!]_t$ and similarly for other levels.

Example 2.1 (Translation, step 1)**.** Up to now, the translation taking into account the continuation-passing style of $a : A, \alpha : A^\perp, b : B \vdash_e e : C$ is simply:

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

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 each time a negation is used. We write this sequent $\vdash [\![\Gamma]\!] \triangleright_{\mathrm{t}} A$ where $\cdot \triangleright_{\mathrm{t}} A$ is defined by induction on t and A:

$$\begin{aligned}[\![\Gamma]\!] \triangleright_{\mathrm{t}} A &= [\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_{\mathrm{E}} A) \to \bot\\ &= [\![\Gamma]\!] \to ([\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_{\mathrm{V}} A) \to \bot) \to \bot \cdots\end{aligned}$$

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

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

Example 2.2 (Translation, step 2)**.** Up to this point, the continuation-and-environment-passing style translation of $a : A, \alpha : A^\perp, b : B \vdash_e e : C$ is:

$$\begin{array}{l}[\![a : A, \alpha : A^\perp, b : B \vdash_e e : C]\!] \\ = \vdash [\![e]\!]_{\mathsf{e}} : [\![a : A, \alpha : A^\perp, b : B]\!] \triangleright_{\mathsf{e}} C \\ = \vdash [\![e]\!]_{\mathsf{e}} : [\![a : A, \alpha : A^\perp, b : B]\!] \to ([\![a : A, \alpha : A^\perp, b : B]\!] \triangleright_{\mathsf{t}} C) \to \perp \\ = \ldots\end{array}$$

where:

$$\begin{array}{l}[\![a : A, \alpha : A^\perp, b : B]\!] \\ \quad = [\![a : A, \alpha : A^\perp]\!],\ b : [\![a : A, \alpha : A^\perp]\!] \triangleright_{\mathsf{t}} B \\ [\![a : A, \alpha : A^\perp]\!] \triangleright_{\mathsf{t}} B \\ \quad = [\![a : A, \alpha : A^\perp]\!] \to ([\![a : A, \alpha : A^\perp]\!] \triangleright_{\mathsf{E}} B) \to \perp = \ldots \\ [\![a : A, \alpha : A^\perp]\!] = [\![a : A]\!],\ \alpha : [\![a : A]\!] \triangleright_{\mathsf{E}} A \\ \quad = [\![a : A]\!],\ \alpha : [\![a : A]\!] \to ([\![a : A]\!] \to \triangleright_{\mathsf{E}} A) \to \perp = \ldots \\ [\![a : A]\!] = a : \varepsilon \triangleright_{\mathsf{t}} A \ = \ a :^{\underline{4}} [\![A]\!]_v\end{array}$$

Step 3 – Extension of the environment The environment-passing style part being settled, it remains to take into account 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 interpretation, this was reflected by the compatibility of realizers with any environment extension [26].

For this purpose, we use as a type system an adaptation of System $F_{<:}$ [7] extended with stores, defined as lists of assignations $[x := t]$. *Store types*, denoted by Υ, are defined as a list of types of the form $(x : A)$ where x is a name and A is a (usual) type. Store types admit a subtyping notion $\Upsilon' <: \Upsilon$ to express that Υ' is an extension of Υ. This corresponds to the following refinement of the definition of $[\![\Gamma]\!] \triangleright_{\mathsf{t}} A$:

$$\begin{array}{l}[\![\Gamma]\!] \triangleright_{\mathsf{t}} A = \forall \Upsilon <: [\![\Gamma]\!].\Upsilon \to (\Upsilon \triangleright_{\mathsf{E}} A) \to \perp \\ \quad = \forall \Upsilon <: [\![\Gamma]\!].\Upsilon \to (\forall \Upsilon' <: \Upsilon.\Upsilon' \to \Upsilon' \triangleright_{\mathsf{V}} A \to \perp) \to \perp \\ \quad = \ldots\end{array}$$

Such a quantification is reminiscent of Kripke forcing [16]: thinking of store types Υ as *worlds*, the *accessible worlds* from Υ 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:

$$\begin{array}{l}\Upsilon \triangleright_{\mathsf{v}} T \to U \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{t}} T) \to (Y \triangleright_{\mathsf{E}} U) \to \perp \\ \omega \Vdash A \Rightarrow B \triangleq \forall \omega' \geq \omega.\ \omega' \Vdash A \ \Rightarrow \ \omega' \Vdash B\end{array}$$

Example 2.3 (Translation, step 3)**.** The translation, now taking into account store extensions, of $a : A, \alpha : A^\perp, b : B \vdash_e e : C$ becomes:

$$\begin{array}{l}[\![a : A, \alpha : A^\perp, b : B \vdash_e e : C]\!] \\ \quad = \vdash [\![e]\!]_{\mathsf{e}} : [\![a : A, \alpha : A^\perp, b : B]\!] \triangleright_{\mathsf{e}} C \\ \quad = \vdash [\![e]\!]_{\mathsf{e}} : \forall \Upsilon <: [\![a : A, \alpha : A^\perp, b : B]\!].\Upsilon \to (\Upsilon \triangleright_{\mathsf{t}} C) \to \perp \\ \quad = \ldots\end{array}$$

where:

$$\begin{array}{l}[\![a : A, \alpha : A^\perp, b : B]\!] = [\![a : A, \alpha : A^\perp]\!],\ b : [\![a : A, \alpha : A^\perp]\!] \triangleright_{\mathsf{t}} B \\ [\![a : A, \alpha : A^\perp]\!] \triangleright_{\mathsf{t}} B \\ \quad = \forall \Upsilon <: [\![a : A, \alpha : A^\perp]\!].\Upsilon \to (\Upsilon \triangleright_{\mathsf{E}} B) \to \perp = \ldots \\ [\![a : A, \alpha : A^\perp]\!] = [\![a : A]\!],\ \alpha : [\![a : A]\!] \triangleright_{\mathsf{E}} A \\ \quad = [\![a : A]\!],\ \alpha : \forall \Upsilon <: [\![a : A]\!].\Upsilon \to (\Upsilon \triangleright_{\mathsf{V}} A) \to \perp = \ldots \\ [\![a : A]\!] = a : \varepsilon \triangleright_{\mathsf{t}} A \ = \ a : \forall \Upsilon.\Upsilon \to (\Upsilon \triangleright_{\mathsf{E}} A) \to \perp\end{array}$$

Step 4 – Explicit coercions The only remaining step is to take into account de Bruijn levels both inside the source and target languages. In the target language, stores will then simply be defined as lists of terms (that is $[t, u, \ldots]$ rather than $[x := t, y := u, \ldots]$) while store types will simply be lists of types (*i.e.* $A, A^\perp, \ldots$ instead of $a : A, \alpha : A^\perp, \ldots$). Interestingly, this requires giving computational content to the subtyping relation $\Upsilon' <: \Upsilon$ through explicit coercions $\sigma : \Upsilon' <: \Upsilon$ mapping each type in Υ to the corresponding type in Υ'. In other words, considering a store $\tau' : \Upsilon'$ extending a store $\tau : \Upsilon$, a coercion $\sigma : \Upsilon' <: \Upsilon$ indicates where each element of τ can be found in τ'.

Remark 1. *Looking carefully at the first three steps, we observe that the translation of a type A at level t is defined by applying a transformation on its translation at level E, which is itself defined as the same transformation applied to the translation at level V and so on. Each step precisely consisted in refining this transformation. Formally, if $\mathcal{F}$ is a (meta) function taking a store type Υ and returning a type, let us define:*

$$\Box\mathcal{F} \ \triangleq \ \Upsilon \mapsto \forall \Upsilon' <: \Upsilon.\Upsilon' \to (\mathcal{F}\Upsilon') \to \perp$$

which is also a function mapping store types Υ to types in the target of the translation. Then, writing $\cdot \triangleright_\iota A$ for the function associating to any store type Υ the type $t \triangleright_\iota A$ (for ι a level among $e, t, E, \ldots$), the definition of the translation can be simply expressed by:

$$\cdot \triangleright_{\mathsf{t}} A = \Box(\cdot \triangleright_{\mathsf{E}} A) = \Box(\Box(\cdot \triangleright_{\mathsf{V}} A)) = \ldots$$

Following this observation, the previous steps are successive refinements of the definition of $\Box\mathcal{F}$:

- *at step 1, $\Box\mathcal{F}$ is $\Upsilon \mapsto (\mathcal{F}\ \Upsilon) \to \perp$,*
- *at step 2, $\Box\mathcal{F}$ is $\Upsilon \mapsto \Upsilon \to \mathcal{F}\ \Upsilon \to \perp$,*
- *at step 3, $\Box\mathcal{F}$ is $\Upsilon \mapsto \forall \Upsilon' <: \Upsilon.\Upsilon' \to (\mathcal{F}\Upsilon') \to \perp$.*

3 F_Υ: a calculus of expandable stores

3.1 Core calculus

The elements necessary to properly define a continuation-and-environment passing style translation being now identified, we can formally introduce the corresponding calculus, which we dub System F_Υ. Each of the translations that we

λ-calculus

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

Stores

$$\frac{}{\Gamma \vdash [] : \emptyset \triangleright_\tau \emptyset}(\emptyset) \qquad \frac{\Gamma \vdash t : \Upsilon_0 \blacktriangleright T}{\Gamma \vdash [t] : \Upsilon_0 \triangleright_\tau T}([t]) \qquad \frac{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \qquad \Gamma \vdash \tau' : (\Upsilon_0; \Upsilon) \triangleright_\tau \Upsilon'}{\Gamma \vdash \tau; \tau' : \Upsilon_0 \triangleright_\tau \Upsilon; \Upsilon'}(\tau;\tau') \qquad \frac{\Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0 \quad \Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon}{\Gamma \vdash \uparrow^\sigma_\Upsilon \tau : \Upsilon_1 \triangleright_\tau \Upsilon}(\uparrow\tau)$$

Coercions

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

$$\frac{\Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash \sigma^{+\Upsilon} : (\Upsilon_1; \Upsilon) <: (\Upsilon_0; \Upsilon)}(<:^\Upsilon_+) \qquad \frac{\Gamma \vdash \sigma : \Upsilon' <: \Upsilon}{\Gamma \vdash \Uparrow \sigma : (\Upsilon', F) <: \Upsilon}(<:_\Uparrow) \qquad \frac{\Gamma \vdash \sigma : \Upsilon' <: \Upsilon}{\Gamma \vdash \Uparrow_{\Upsilon''} \sigma : (\Upsilon'; \Upsilon'') <: \Upsilon}(<:^\Upsilon_\Uparrow)$$

Core

$$\frac{(\delta : \Upsilon_0 \triangleright_\tau \Upsilon) \in \Gamma}{\Gamma \vdash \delta : \Upsilon_0 \triangleright_\tau \Upsilon}(\tau_{\mathrm{ax}}) \qquad \frac{\Gamma, \delta : \Upsilon_0 \triangleright_\tau \Upsilon \vdash t : B}{\Gamma \vdash \lambda\delta.t : \Upsilon_0 \triangleright_\tau \Upsilon \to B}(\tau_I) \qquad \frac{\Gamma \vdash t : \Upsilon_0 \triangleright_\tau \Upsilon \to B \quad \Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon}{\Gamma \vdash t\,\tau : B}(\tau_E) \qquad \frac{(s : Y <: \Upsilon) \in \Gamma}{\Gamma \vdash s : Y <: \Upsilon}(<:_{\mathrm{s}})$$

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

$$\frac{\Gamma \vdash t : B \quad A \equiv_\Upsilon B}{\Gamma \vdash t : A}(\equiv)$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \Upsilon' \qquad \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \qquad \Upsilon = [\![\Gamma_0]\!], F, [\![\Gamma_1]\!] \qquad |\Gamma_0| = n \\ \Gamma', s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \blacktriangleright F, s_1 : (Y_0, F; Y_1 <: \Upsilon), \delta_1 : (Y_0, F) \triangleright_\tau Y_1 \vdash t : A'\end{array}}{\Gamma \vdash \mathtt{split}\ \tau\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon' <: \Upsilon\ \mathtt{as}\,(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ t : A}(\mathtt{split})$$

where $\Gamma' = \Gamma[Y_0, F; Y_1/\Upsilon']$ *and* $A' = A[Y_0, F; Y_1/\Upsilon']$

Figure 6. F_Υ: Type system

will introduce afterwards will rely on a target calculus defined as a particular instance of this system. The structure of these translations can be divided in three blocks:

1. A **source calculus** and its type system. For the moment, we will only consider simply-typed λ-calculi[10] to ease notations, we shall explain afterwards in Section 4.3 how this scales to System F.

2. A syntax for **stores** and **coercions**. In this paper, we will simply consider lists and list inclusions, but one could also contemplate the possibility of using trees or different data structures. The type of a store will always be a list of types from the source calculus, while the way these types are translated will be a parameter of the target calculus depending on the chosen translation. In other words, the interpretation of a store of type $A, A^\perp, B$ will depend on whether we consider a call-by-need or a call-by-value translation. As for the type of a coercion, it will simply be expressed as the inclusion relation for the corresponding lists of types.

3. The **target calculus**, which will be an instance of F_Υ. This calculus should be as expressive as the source calculus (here, this will only mean to have the same constants) and should contain enough structure to manipulate stores and express extensions of stores. Our calculus will be parameterized by the base cases for translation of source types together with the translation of types in the store.

We are now ready to introduce the syntax of F_Υ. As we explained, we will focus on the case of a simply typed source calculus, whose types are defined by:

$$\textit{Source types} \qquad A ::= X \mid A \to B \qquad\qquad F ::= A \mid A^\perp$$

As explained above, a store τ is simply given as a list whose type, written Υ, is defined as the list of the types of the elements τ provides. Observe that if we consider the concatenation of such two stores $\tau_0; \tau_1$ of type $\Upsilon_0; \Upsilon_1$, thinking of them as substitutions, τ_1 provides elements for Υ_1 only if it is given with a prefix providing Υ_0 (see also the Step 2 described earlier). We denote its type with $\Upsilon_0 \triangleright_\tau \Upsilon_1$, expressing the fact that it is well-typed only when concatenated with a store of type $\emptyset \triangleright_\tau \Upsilon_0$ (the type of a "self-sufficient" store, which we will abbreviate by Υ_0). Obviously, such a store should also be compatible with any extension of Υ_0, which justifies the introduction of the constructor $\uparrow^\sigma_{\Upsilon_1} \tau$ where σ is of type $\Upsilon_0' <: \Upsilon_0$ and the resulting store is of type $\Upsilon_0' \triangleright_\tau \Upsilon_1$.

Coercions are simply defined through the natural structure witnessing list inclusion: σ^+ (resp. $\Uparrow \sigma$) witnesses the fact that Υ, F (resp. Υ) is included in Υ', F provided that σ witnesses that Υ is included in Υ'. Having in mind that list inclusion corresponds to a subtyping relation for stores, we denote[11] it by $\Upsilon' <: \Upsilon$. We denote by $\sigma' \circ \sigma$ the composition of two coercions, and by $\sigma^{+\Upsilon}$ (resp. $\Uparrow_\Upsilon \sigma$) the twin of the previous coercions witnessing the addition of a list of type Υ.

Finally, System F_Υ terms are defined as the usual λ-terms (with constants, to match the source calculus expressiveness) extended with abstractions (and the dual applications) over stores and coercions as well as a `split` instruction to be able

[10] In fact, variants of the $\bar{\lambda}\mu\tilde{\mu}$-calculus with explicit stores, but which would naturally define an Intermediate Representation for the corresponding λ-calculi in the compilation chain.

[11] This may read "Υ' *is more precise than* Υ *as a list*" since it contains all its elements.

to split a given store using a coercion. The resulting syntax is given by the following grammar:

$$\begin{array}{lll} \textit{Store type} & \Upsilon ::= Y \mid \emptyset \mid \Upsilon, F \mid \Upsilon; \Upsilon' \\ \textit{Stores} & \tau ::= \delta \mid [] \mid [t] \mid \tau; \tau' \mid \Uparrow^{\sigma}_{\Upsilon} \tau \\ \textit{Coercions} & \sigma ::= s \mid \varepsilon \mid \sigma^{+} \mid \Uparrow \sigma \mid \sigma \circ \sigma' \mid \sigma^{+_\Upsilon} \mid \Uparrow_{\Upsilon} \sigma \end{array}$$

$$\begin{array}{lll} \textit{Contexts} & \Gamma ::= \cdot \mid \Gamma, s : \Upsilon' <: \Upsilon \mid \Gamma, \delta : \Upsilon \triangleright_{\tau} \Upsilon' \mid \Gamma, x : T \\ \textit{Types} & T ::= X \mid T \to U \mid \forall Y.T \mid \Upsilon' <: \Upsilon \to T \mid \Upsilon \triangleright_{\tau} \Upsilon' \to T \\ \textit{Terms} & t ::= \boldsymbol{k} \mid x \mid \lambda x.t \mid t\, u \mid \lambda s.t \mid t\, \sigma \mid \lambda \delta.t \mid t\, \tau \\ & \mid \lambda Y.t \mid t\, \Upsilon \mid \mathtt{split}\ \tau\ \mathtt{at}\ n\ \begin{array}{l}\mathtt{along}\ \sigma : \Upsilon' <: \Upsilon \\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\end{array}\ \mathtt{in}\ t \end{array}$$

As for the typing rules, there are given in Figure 6 where $\mathcal{S}$ is a signature for constants (which will be chosen in each instance so that constants types match the types expected from the source calculus), where $\equiv_{\Upsilon}$ is the congruent-transitive-symmetric-reflexive closure of the reduction $\to_{\Upsilon}$ (defined in the next section) and where $\Upsilon \blacktriangleright F$ is a parameter of the translation. Intuitively, its definition will specify how elements of the global environment are translated (*e.g.*, in call-by-need the environment contains potentially unevaluated terms[12], which will be translated as such). Since our syntax implicitly assume that stores of types $\Upsilon_0 \triangleright_{\tau} \Upsilon$ can be lifted to match extensions of Υ_0, for our system to be sound, we need to assume that terms in the store can be lifted as well. Formally, we require that $\Upsilon \blacktriangleright F$ is of the shape[13]:

$$(\mathrm{Eq}_{\blacktriangleright}) \qquad \Upsilon \blacktriangleright F \equiv \forall Y <: \Upsilon.\mathcal{F}(Y, F)$$

for some $\mathcal{F}$, where we write $\forall Y <: \Upsilon.T$ as an abbreviation for $\forall Y.Y <: \Upsilon \to T$. All the typing rules are quite intuitive, but for the (split) rule, which requires special care. For a term:

$$\mathtt{split}\ \tau\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon' <: \Upsilon\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ t$$

to be well-typed, we need to ensure that Υ is of the shape $[\![\Gamma_0]\!], F, [\![\Gamma_1]\!]$ where we abuse the notation $[\![\Gamma_0]\!]$ to express that we require a concrete list of types (*i.e.* without store variable Y). Besides, to allow the term t to contain coercions referring to the resulting split stores, we need to explicitly name their types, hence[14] the Y_0 and Y_1.

Before we present the operational semantics of the calculus, observe that we introduced enough structure to define terms whose type will match the transformation $\Box$ evoked earlier (see Remark 1). Remember that we defined the formula $\Box\mathcal{F}\Upsilon$ as $\forall Y <: \Upsilon.\emptyset \triangleright_{\tau} Y \to (\mathcal{F}Y) \to \bot$ for any Υ. We have:

Lemma 3.1. *Assume that for any Υ, $\mathcal{F}\Upsilon$ is a type. The following rules are admissible:*

$$\frac{\Gamma, s : Y <: \Upsilon, \delta : \emptyset \triangleright_{\tau} Y, x : \mathcal{F}Y \vdash t : \bot}{\Gamma \vdash \lambda s \delta x.t : \Box\mathcal{F}\Upsilon}\ (\Box_I)$$

$$\frac{\Gamma \vdash t : \Box\mathcal{F}\Upsilon \quad \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \quad \Gamma \vdash \tau : \emptyset \triangleright_{\tau} \Upsilon' \quad \Gamma \vdash u : \mathcal{F}\Upsilon'}{\Gamma \vdash t\, \sigma\, \tau\, u : \bot}\ (\Box_E)$$

[12] As opposed to values in call-by-value.

[13] Where we use the notation $\forall Y <: \Upsilon.T$ to abbreviate $\forall Y.Y <: \Upsilon \to T$.

[14] As the reader will observe in the typing rules, we also need to identify the current store type (Υ') with its cutting ($Y_0, F; Y_1$). See also Remark 2.

As we shall see in the next sections, we will mainly use these "packed" abstractions and applications to define the different translations (observe that $\Box\mathcal{F}\Upsilon$ satisfies ($\mathrm{Eq}_{\blacktriangleright}$)). This somehow suggests the use of lists for stores and the corresponding coercions is more a matter of implementation choices than a crucial point in the definition of F_{Υ}. Actually, we could have presented F_{Υ} directly through these rules, but we choose to stick to a somewhat more atomic presentation.

3.2 Reduction rules

We shall now define the reduction rules of System F_{Υ}. If most of them are straightforward, some special care has to be given to define the reduction of terms of the shape $\mathtt{split}\ \tau\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon' <: [\![\Gamma]\!]\ \mathtt{as}\ \ldots\ \mathtt{in}\ t$. To reduce such a term, we will need to use the coercion σ to split the store τ correctly. In particular, we need σ (and τ) to be in *normal form*, that is that it contains neither variables nor compositions of coercions:

$$\begin{array}{lll} \textbf{Normal forms} & \tau_n ::= [] \mid \tau_n[t] & \sigma_n ::= \varepsilon \mid \sigma_n^{+} \mid \Uparrow \sigma_n \\ & \Upsilon_n ::= \emptyset \mid \Upsilon_n, F \end{array}$$

As shown by the next lemma, a coercion in normal forms always allows us to split a store type and the coercion itself:

Lemma 3.2. *If $\Gamma \vdash \sigma : \Upsilon <: [\![\Gamma_0]\!], A; [\![\Gamma_1]\!]$ where σ is a coercion in normal form then there exists $\Upsilon_0, \Upsilon_1, \sigma_0, \sigma_1$ such that:*

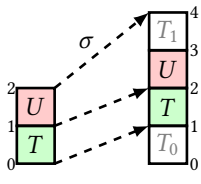
1. $\Gamma \vdash \sigma_0 : \Upsilon_0 <: [\![\Gamma_0]\!]$
2. $\Gamma \vdash \sigma_1 : \Upsilon_1 <: [\![\Gamma_1]\!]$
3. $\Upsilon = \Upsilon_0, A; \Upsilon_1$
4. $\sigma = \sigma_0^{+}; \sigma_1$

where $\sigma_0; \sigma_1$ is the coercion σ_1 where σ_0 replaces ε.

Alternatively, a coercion $\sigma : \Upsilon' <: \Upsilon$ in normal form can actually be identified with a partial monotone function $[\![\sigma]\!]_{\mathbb{N}}$ from $[0, |\Upsilon|]$ to $[0, |\Upsilon'|]$ which intuitively maps the index of every type in Υ to its corresponding index in Υ' (see Appendix D.1).

Example 3.3. As an example, for any types T, U, T_0, T_1, if we write σ to denote the coercion $\Uparrow((\Uparrow \varepsilon)^{++})$ which is in normal form, we have:

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



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

Similarly, it is easy to see that if a store τ is in normal form and is of type $[\![\Gamma_0]\!], A; [\![\Gamma_1]\!]$, then we can divide it as expected into $\tau = \tau_0[u]; \tau_1$. This justifies the reduction rule for split:

$$\begin{array}{c} \mathtt{split}\ \tau_0[u]; \tau_1\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon_0, A, \Upsilon_1 <: [\![\Gamma_0]\!], A, [\![\Gamma_1]\!] \\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ t \\ \to \\ t[\Upsilon_0/Y_0][\sigma_0/s_0][\tau_0/\delta_0][u/x][\Upsilon_1/Y_1][\tau_1/\delta_1][\sigma_1/s_1] \end{array}$$

```
Parameter Ty:Type.

Inductive hlist:=
 |hnil : hlist
 |hcons : ∀ A, A → hlist → hlist.

Definition storeT := list Ty.

Inductive <: : storeT → storeT → Type:=
 | empty : nil <: nil
 | lift : ∀ Υ' Υ A, Υ'<: Υ → [Υ',A] <: Υ
 | plus : ∀ Υ' Υ A, Υ'<: Υ → [Υ',A] <: [Υ,A].
```

Coercions

```
Parameter ▶ : storeT → Ty → Type.

Context (F: storeT → Ty → Type)
        ( tr_sound:∀ Υ A, (Υ ▶ A) = ∀ Z, Z<:Υ → F Z A).

Definition store := hlist.

Inductive typedStore:store →storeT → storeT → Type :=
 | nilT : ∀ Y, typedStore hnil Y nil
 | consT : ∀ (A:Ty) (τ:store) (Υ0, Υ1:storeT) (t:Υ0; Υ1 ▶ A),
    typedStore τ Υ0 Υ1→ typedStore τ[t] Υ0 Υ1,A.
```

Stores and store types

Figure 7. Shallow embedding of $F_\Upsilon$into Coq

where $n = |\Gamma_0|, |\tau_0| = |\Upsilon_0|$ and $\Upsilon_0, \Upsilon_1, \sigma_0, \sigma_1$ are as in Lemma 3.2 (i.e. $|\Upsilon_0| = [\![\sigma]\!](n)$, $\sigma_0 : \Upsilon_0 <: \Gamma_0$, etc.).

The rest of the reduction rules are easier and of four kinds:
a) Reductions that normalize coercions, stores and store types as much as possible:

$$\begin{array}{ll} \sigma_1^+ \circ \Uparrow \sigma_0 \to_\sigma \Uparrow(\sigma_1 \circ \sigma_0) & \varepsilon^{+\Upsilon} \circ \sigma \to_\sigma \sigma \\ \sigma_1^+ \circ \sigma_0^+ \to_\sigma (\sigma_1 \circ \sigma_0)^+ & \sigma \circ \varepsilon^{+\Upsilon} \to_\sigma \sigma \\ \Uparrow \sigma_1 \circ \sigma_0 \to_\sigma \Uparrow(\sigma_1 \circ \sigma_0) & \tau; (\tau'[t]) \to_\tau (\tau; \tau')[t] \\ \Uparrow_{\Upsilon,F} \sigma \to_\sigma \Uparrow \Uparrow_\Upsilon \sigma & \uparrow^\sigma_{\Upsilon,F} \tau[t] \to_\tau (\uparrow^\sigma_\Upsilon \tau)[\uparrow^{\sigma^{+\Upsilon}} t] \\ \sigma^{+_{\Upsilon,F}} \to_\sigma (\sigma^{+_\Upsilon})^+ & \Upsilon; (\Upsilon', F) \to_\Upsilon (\Upsilon; \Upsilon'), F \end{array}$$

where $\uparrow^\sigma t \triangleq \lambda Y s.t\ Y\ s \circ \sigma$.

b) Usual β-reduction steps for the different abstractions:

$$\begin{array}{ll} (\lambda x.t)\, u \to t[u/x] & (\lambda s.t)\, \sigma \to t[\sigma/s] \\ (\lambda Y.t)\, \Upsilon \to t[\Upsilon/x] & (\lambda \delta.t)\, \tau \to t[\tau/\delta] \end{array}$$

c) Contextual rules for store types in stores and coercions:

$$U_\sigma[\Upsilon] \overset{\text{if } \Upsilon \to_\Upsilon \Upsilon'}{\longrightarrow_\sigma} U_\sigma[\Upsilon'] \qquad U_\tau[\Upsilon] \overset{\text{if } \Upsilon \to_\Upsilon \Upsilon'}{\longrightarrow_\tau} U_\tau[\Upsilon']$$

where $U_\sigma[] ::= \sigma^{+[]} \,|\Uparrow_{[]}\sigma$ and $U_\tau[] ::= \uparrow^\sigma_{[]} \tau]$
d) Contextual rules for stores and coercions in split:

$$C[\sigma] \overset{\text{if } \sigma \to_\sigma \sigma'}{\longrightarrow} C[\sigma'] \qquad D[\tau] \overset{\text{if } \tau \to_\tau \tau'}{\longrightarrow} D[\tau']$$

where contexts are defined by:

$$\begin{array}{l} C[] ::= \texttt{split}\ [] \ \texttt{at}\ n\ \texttt{along}\ \sigma : \Upsilon' <: \Upsilon\ \texttt{as}\ \ldots\ \texttt{in}\ t \\ D[] ::= \texttt{split}\ \tau\ \texttt{at}\ n\ \texttt{along}\ [] : \Upsilon' <: \Upsilon\ \texttt{as}\ \ldots\ \texttt{in}\ t \end{array}$$

Theorem 3.4 (Subject reduction). *If* $\Gamma \vdash t : T$ *and* $t \to t'$ *then* $\Gamma \vdash t' : T$.

Proof. See Appendix D.2. □

Theorem 3.5 (Normalization). *Typed terms normalize.*

Proof. The proof is done by means of a realizability interpretation, see Appendix D.3. □

3.3 Expressiveness

Before introducing the different translations for which we will us F_Υ as target calculus, let us briefly discuss the expressiveness of the system. To be more precise, we are interested in the expressiveness that is required to define F_Υ (independently of the source calculus). As the reader would have observed, not only do we need to manipulate lists of types and their size, but our definition of coercions involves a form of dependent types (as in $\sigma^{+\Upsilon}$ and $\Uparrow_\Upsilon \sigma$). Even if we could have thought to different presentations, it seems that there is a minimal form of dependent types (namely to have access to the size of lists) that we cannot get rid of. In particular, it seems very unlikely that our calculus could be embed into system F or Cardelli's system $F_{<:}$.

In turn, it is clear that various dependent type theories are expressive enough to define F_Υ. As an example, in Figure 7 we give an overview of how F_Υ can be expressed in Coq through a shallow embedding. We give more details in Appendix E[15].

4 Implementation for simply typed calculi

We are now equipped to define typed continuation-and-environment-passing style translations to system F_Υ. These translations exactly follow the intuitions outlined in Section 2.1. 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 H.

[15]The Coq development illustrating these ideas is available at: https://gitlab.com/emiquey/fupsilon.

$$[\![\Gamma_i, \Gamma' \vdash \lambda x_i.t : A \to B]\!]_{\mathsf{v}}\ \Upsilon\ \sigma\ \tau\ u\ E \triangleq [\![t]\!]_{\mathsf{t}}\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[u]\ \uparrow^{\Uparrow \mathsf{id}_\Upsilon} E \qquad [\![\boldsymbol{k}]\!]_{\mathsf{v}} \triangleq \boldsymbol{k}$$

$$[\![t \cdot E]\!]_{\mathsf{F}}\ \Upsilon\ \ \sigma\ \tau\ v \triangleq v\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![t]\!]_{\mathsf{t}})\ (\uparrow^\sigma [\![E]\!]_{\mathsf{E}}) \qquad [\![\boldsymbol{\kappa}]\!]_{\mathsf{F}} \triangleq \boldsymbol{\kappa}$$

$$[\![v]\!]_{\mathsf{V}}\ \Upsilon\ \ \sigma\ \tau\ F \triangleq F\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![v]\!]_{\mathsf{v}})$$

$$[\![\Gamma_0, x_i : A, \Gamma_1 \vdash x_i : A]\!]_{\mathsf{V}}\ \Upsilon\ \sigma\ \tau\ F \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ (\sigma : \Upsilon <: [\![\Gamma_0]\!], A; [\![\Gamma_1]\!])\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\quad \textit{where } \sigma' = (s^+)^{+_{Y_1}}$$
$$\mathtt{in}\ x\ Y_0\ \mathsf{id}_{Y_0}\ \delta_0\ (\lambda Y s \delta V.V\ (Y, A; Y_1)\ (\Uparrow\!\!\!\Uparrow_{Y_1} \mathsf{id}_Y)\ (\delta[\uparrow^{\mathsf{t}} V]; \uparrow^{s^+} \delta_1)\ (\uparrow^{\sigma'} F)) \quad \textit{and } \uparrow^{\mathsf{t}} V = \lambda Y s \delta E.E\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s V)$$

$$[\![\alpha_i : A^\perp]\!]_{\mathsf{E}}\ \Upsilon\ \sigma\ \tau\ V \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ x\ \Upsilon\ (\Uparrow\!\!\!\Uparrow_{Y_1} \mathsf{id}_{Y_0})\ \tau\ V$$

$$[\![\Gamma_i, \Gamma'' \vdash \tilde{\mu}[x_i].\langle x_i \parallel F\rangle \tau' : A^\perp]\!]_{\mathsf{E}}\ \Upsilon\ \sigma\ \tau\ V\ \triangleq V\ (\Upsilon, A; [\![\Gamma']\!])\ \sigma_V\ (\tau[\uparrow^{\mathsf{t}} V]; \uparrow^{\sigma_\tau} [\![\tau']\!]_\tau)\ (\uparrow^{\sigma_\tau^{+[\![\Gamma']\!]}} [\![F]\!]_{\mathsf{F}}) \quad \textit{where}^\dagger \begin{cases} \sigma_\tau = (s \circ \Uparrow_{[\![\Gamma'']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+ \\ \sigma_V = \Uparrow_{A;[\![\Gamma']\!]} \mathsf{id}_\Upsilon \end{cases}$$

$$[\![V]\!]_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ E \triangleq E\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![V]\!]_{\mathsf{V}}) \qquad [\![\Gamma_i, \Gamma' \vdash \mu\alpha_i.c : A]\!]_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ E \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon, A^\perp)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[E]$$

$$[\![E]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ t \triangleq t\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![E]\!]_{\mathsf{E}}) \qquad [\![\Gamma_i, \Gamma' \vdash \tilde{\mu}x_i.c : A^\perp]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ t \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[t]$$

$$[\![\langle t \parallel e\rangle]\!]_{\mathsf{c}}\ \Upsilon\ \sigma\ \tau \triangleq [\![e]\!]_{\mathsf{e}}\ \ \Upsilon\ \sigma\ \tau\ (\uparrow^\sigma [\![t]\!]_{\mathsf{t}}) \qquad [\![c\tau']\!]_{\mathsf{l}}\ \Upsilon\ \sigma\ \tau \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon; \Gamma')\ (\sigma^{+[\![\Gamma']\!]})\ (\tau; \uparrow^\sigma [\![\tau']\!]_\tau) \qquad \textit{where}^\dagger\ \tau' : \Gamma'$$

$$[\![\tau_0[x_i := t]]\!]_\tau \triangleq [\![\tau_0]\!]_\tau[[\![t]\!]_{\mathsf{t}}] \qquad [\![\varepsilon]\!]_\tau \triangleq \varepsilon \qquad [\![\tau_0[\alpha_i := E]]\!]_\tau \triangleq [\![\tau_0]\!]_\tau[[\![E]\!]_{\mathsf{E}}]$$

where $\mathsf{id}_\Upsilon = \varepsilon^{+_\Upsilon}$, $\sigma_{\Gamma,\Gamma'} \triangleq (\sigma \circ \Uparrow_{[\![\Gamma']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+$ and Γ_i indicates that $|\Gamma| = i$ $\quad$ $^\dagger\tau' : \Gamma'$ in the source of the translation

Figure 8. Call-by-need translation of terms

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

4.1.1 Translation of terms

We can now take advantage of the features of system F_Υ to type Ariola *et al.*'s untyped translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus (given in Figure 14). This translation was obtained by refining the reduction system (see Figure 12) 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 $\tau[t]; \tau'$, t has its levels coherent with its prefix τ, and ignores its suffix τ'. In terms of types, if $\tau[t]$ is of type Υ, A then t will be of a type $\Upsilon \triangleright_{\mathsf{t}} A$.
2. The continuations/terms that are passed with a store are always consistent with it, that is they do not need to be lifted and their types always match the type of the store.

Let us spend a few lines to explain the definitions of two cases, namely $[\![x_i]\!]_{\mathsf{V}}$ and $[\![\tilde{\mu}[x_i].\langle x \parallel F\rangle \tau']\!]_{\mathsf{E}}$. In the untyped translation (see Appendix B.1), we have:

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

Let us first focus on the $[\![\tilde{\mu}[x_i].\langle x \parallel F\rangle \tau']\!]_{\mathsf{E}}$. In the named version, its translation is a term which waits for a store τ and a value V, then forms a store looking like $\tau[x := [\![V]\!]]\tau'$ and passes it to V with the continuation $[\![F]\!]_{\mathsf{F}}$. Now, when using de Bruijn levels, the continuation was expecting a store τ_0 of type Υ_0 that τ (the actual store, of type Υ) is extending, hence τ comes together with a coercion $\sigma : \Upsilon <: \Upsilon_0$ witnessing that Υ it is an extension of Υ_0. Let us loosely identify the stores with their types and write $\sigma : \tau <: \tau_0$ to simplify our explanation. The value V has its de Bruijn levels consistent with τ, but $[\![F]\!]_{\mathsf{F}}$ and $[\![\tau']\!]_\tau$ have to be updated. In detail, $[\![\tau']\!]_\tau$ was expecting $\tau_0[\uparrow^{\mathsf{t}} V]$, we thus update it with σ^+ which witnesses $\tau[\uparrow^{\mathsf{t}} V] <: \tau_0[\uparrow^{\mathsf{t}} V]$. On the other hand, F was waiting for $\tau_0[\uparrow^{\mathsf{t}} V]; \tau'$, we thus need to update it with $\sigma' = (\sigma^+)^{+[\![\tau']\!]}$. Finally, we need to give to V a coercion witnessing the extension of τ into $\tau[\uparrow^{\mathsf{t}} V]_t; [\![\tau']\!]_\tau$, that is to say $\Uparrow_{[\![\tau']\!]} \mathsf{id}_\tau$. In the end, we obtain the following definition:

$$[\![\tilde{\mu}[x_i].\langle x_i \parallel F\rangle \tau']\!]_{\mathsf{E}}\ \Upsilon\ \sigma\ \tau\ V \triangleq$$
$$V\ (\Upsilon, A; [\![\Gamma']\!])\ (\Uparrow_{[\![\Gamma']\!]} \mathsf{id}_\Upsilon)\ (\tau[\uparrow^{\mathsf{t}} V]_t; \uparrow^{\sigma^+} [\![\tau']\!]_\tau)\ \uparrow^{\sigma'} [\![F]\!]_{\mathsf{F}}$$

where $\mathsf{id}_\Upsilon = \varepsilon^{+_\Upsilon}$, $\sigma' =\Uparrow_{[\![\Gamma']\!]} \mathsf{id}_\Upsilon$ and $\tau' : \Gamma'$ in the source calculus.

As for $[\![x_i]\!]_{\mathsf{V}}$, it is a term waiting for a coercion σ and a store τ, which it will split at $[\![\sigma]\!]_{\mathbb{N}}(i)$ as τ_0, t, τ_1 to execute the term t with its prefix τ_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 8, where we assume that for each constant $\boldsymbol{k}$ of type T (resp. co-constant $\boldsymbol{\kappa}$ of type $T^\perp$) of the source system, we have a constant of

$$\begin{array}{l|l|l}
[\![\Gamma \vdash_e e : T^\perp]\!] \triangleq \vdash [\![e]\!]_e : [\![\Gamma]\!] \triangleright_{\mathsf{e}} T & [\![\Gamma \vdash_v v : T]\!] \triangleq \vdash [\![v]\!]_v : [\![\Gamma]\!] \triangleright_{\mathsf{v}} T & [\![\varepsilon]\!] \triangleq \varepsilon \\
[\![\Gamma \vdash_t t : T]\!] \triangleq \vdash [\![t]\!]_t : [\![\Gamma]\!] \triangleright_{\mathsf{t}} T & [\![\Gamma \vdash_c c]\!] \triangleq \vdash [\![c]\!]_c : [\![\Gamma]\!] \triangleright_{\mathsf{c}} \perp & [\![\Gamma, x_i : T]\!] \triangleq [\![\Gamma]\!], T \\
[\![\Gamma \vdash_E E : T^\perp]\!] \triangleq \vdash [\![E]\!]_E : [\![\Gamma]\!] \triangleright_{\mathsf{E}} T & [\![\Gamma \vdash_l l]\!] \triangleq \vdash [\![l]\!]_l : [\![\Gamma]\!] \triangleright_{\mathsf{c}} \perp & [\![\Gamma, \alpha_i : T^\perp]\!] \triangleq [\![\Gamma]\!], T^\perp \\
[\![\Gamma \vdash_V V : T]\!] \triangleq \vdash [\![V]\!]_V : [\![\Gamma]\!] \triangleright_{\mathsf{V}} T & [\![\Gamma \vdash_\tau \tau : \Gamma']\!] \triangleq \vdash [\![\tau]\!]_\tau : [\![\Gamma]\!] \triangleright_\tau [\![\Gamma']\!] &
\end{array}$$

(a) Translation of judgments (b) Translation of contexts

$$\begin{array}{l|l|l}
\Upsilon \triangleright_{\mathsf{c}} T \triangleq \forall Y <: \Upsilon.\, Y \to \perp & \Upsilon \triangleright_{\mathsf{V}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_{\mathsf{F}} T) \to \perp & \Upsilon \blacktriangleright T \triangleq \Upsilon \triangleright_{\mathsf{t}} T \\
\Upsilon \triangleright_{\mathsf{e}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_{\mathsf{t}} T) \to \perp & \Upsilon \triangleright_{\mathsf{F}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_{\mathsf{v}} T) \to \perp & \Upsilon \blacktriangleright T^\perp \triangleq \Upsilon \triangleright_{\mathsf{E}} T \\
\Upsilon \triangleright_{\mathsf{t}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_{\mathsf{E}} T) \to \perp & \Upsilon \triangleright_{\mathsf{v}} X \triangleq X & \\
\Upsilon \triangleright_{\mathsf{E}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_{\mathsf{V}} T) \to \perp & \Upsilon \triangleright_{\mathsf{v}} T \to U \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_{\mathsf{t}} T) \to (Y \triangleright_{\mathsf{E}} U) \to \perp &
\end{array}$$

(c) Translation of types

Figure 9. Call-by-need translation of judgments and types

type T in the signature of the target language that we also denote by $\boldsymbol{k}$ (resp. $\boldsymbol{\kappa}$ of type $T \to \perp$).

4.1.2 Translation of types

Regarding the translation of types, it follows exactly the intuition we presented in Section 2.1, so that we mostly said everything about it already. To summarize the construction, we start by embedding the types and typing contexts of the source calculus thanks to the ι function. A typing context $\Gamma = x_1 : T_1, \dots x_n : T_n$ is then translated into the store type $[\![\Gamma]\!] = T_1, \dots, T_n$. This allows us to translate a sequent *e.g.*, $\Gamma \vdash_t t : T$, into a judgment $\vdash [\![t]\!]_{\mathsf{t}} : [\![\Gamma]\!] \triangleright_{\mathsf{t}} T$. The type $[\![\Gamma]\!] \triangleright_{\mathsf{t}} T$ can be understood as the types of terms translated at level t, which are waiting for any store extending $[\![\Gamma]\!]$ and a continuation at the inferior level (*i.e.* E) for the very same type T but interpreted with the extended store type Y:

$$[\![\Gamma]\!] \triangleright_{\mathsf{t}} T = \forall Y <: [\![\Gamma]\!].\, Y \to (Y \triangleright_{\mathsf{E}} T) \to \perp$$

It is worth noting that the translation $- \triangleright_{\mathsf{t}} -$ is defined internally in system F_Υ, which allows in particular the recursive definition $Y \triangleright_{\mathsf{E}} T$ to make sense (since Y is at the moment an interleaving of types in Γ 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[16].

The resulting translation, which is given in Figure 9, is sound, *i.e.* the provability of a sequent in the source calculus (say $\Gamma \vdash_t t : T$) entails the provability of its translation ($[\![\Gamma \vdash_v v : T]\!]$) in F_Υ. which is the main result of this section:

Theorem 4.1 (Soundness)**.** *The translation is well-typed.*

Proof. The proof is done by induction on typing derivations, and requires a few previous lemmas on the soundness of the constructions $\uparrow_V^{+t}$ and $\uparrow^\sigma t$. The complete proof is given in Appendix F. □

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

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

To emphasize that F_Υ 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 3). A similar translation for the same calculus evaluated in call-by-value is given in Appendix H. We first rephrase its reduction rules to use de Bruijn levels:

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

We spare the reader from the redefinition of a type system using de Bruijn levels, which is fully deducible from the type system 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. 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 the translation of terms. The definition of the translation for terms 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. In particular, the definition ensures that the same invariants about consistency with respect to de Bruijn levels and store extension are maintained.

$$[\![\Gamma_i, \Gamma' \vdash \lambda x_i.t : A \to B]\!]_{\mathsf{V}}\ \Upsilon\ \sigma\ \tau\ u\ E \triangleq [\![t]\!]_t\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[u]\ \uparrow^{\Uparrow \mathsf{id}_\Upsilon} E$$
$$[\![t \cdot E]\!]_{\mathsf{E}}\ \Upsilon\ \sigma\ \tau\ v \triangleq v\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![t]\!]_{\mathsf{t}})\ (\uparrow^\sigma [\![E]\!]_{\mathsf{E}})$$

$$[\![x_i]\!]_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ E \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ (Y_0, \_, \_), x, (Y_1, \_, \_)\ \mathtt{in}\ x\ \Upsilon\ (\Uparrow\!\!\Uparrow_{Y_1} \mathsf{id}_{Y_0})\ \tau\ E$$
$$[\![\Gamma_i, \Gamma' \vdash \tilde{\mu} x_i.c : A^\perp]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ t \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[t]$$

(a) Translation of terms (excerpt)

$$[\![\Gamma \vdash_e e : T^\perp]\!] \triangleq \vdash [\![e]\!]_e : [\![\Gamma]\!] \triangleright_{\mathsf{e}} T \qquad [\![\Gamma \vdash_t t : T]\!] \triangleq \vdash [\![t]\!]_t : [\![\Gamma]\!] \triangleright_{\mathsf{t}} T$$
$$[\![\Gamma \vdash_E E : T^\perp]\!] \triangleq \vdash [\![E]\!]_E : [\![\Gamma]\!] \triangleright_{\mathsf{E}} T \qquad [\![\Gamma \vdash_V V : T]\!] \triangleq \vdash [\![V]\!]_V : [\![\Gamma]\!] \triangleright_{\mathsf{V}} T$$

$$\Upsilon \triangleright_{\mathsf{e}} T \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{t}} T) \to \bot \qquad \Upsilon \triangleright_{\mathsf{t}} T \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{E}} T) \to \bot$$
$$\Upsilon \triangleright_{\mathsf{E}} T \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{V}} T) \to \bot \qquad \Upsilon \triangleright_{\mathsf{V}} T \to U \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{t}} T) \to (Y \triangleright_{\mathsf{E}} U) \to \bot$$

$$\Upsilon \blacktriangleright T \triangleq \Upsilon \triangleright_{\mathsf{t}} T \qquad \Upsilon \blacktriangleright T^\perp \triangleq \Upsilon \triangleright_{\mathsf{E}} T$$

(b) Translation of types and judgments (excerpt)

Figure 10. Call-by-name continuation-and-environment-passing style translation (excerpt)

As for the translation of types and judgments, we hope that it should now look trivial to the reader: it follows the exact same guidelines than in the call-by-need case, except that it now only has four levels. It does not come as a surprise that the two translations are so so close: in both cases, terms and contexts are stored at the same syntactic levels in the environments (t and E). The main difference lies in the fact that in call-by-name, terms remains unevaluated in the store and thus there is no need for an extra layer of alternation to handle their (shared) evaluation. Both translations are given in Figure 10, and again, we have:

Theorem 4.2. *The translation is well-typed.*

Proof. The proof is similar (and easier) than the proof in the call-by-need case, by induction on typing derivations. See Appendix G. □

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 $[\![t]\!]_{\mathsf{t}}$ is evaluated, corresponding to other branches of the global typing derivation. As a consequence, the store may contain arbitrarily more elements at that time (see Example G.1).

Example 4.3. Consider a term $x : A, y : B \vdash_t u : C$, through the translation we will thus have $\vdash [\![u]\!]_{\mathsf{t}} : A, B \triangleright_{\mathsf{t}} C \to D$. 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 $\tau_i : \vec{U_i}$) 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 $\vec{U_0}, A, \vec{U_1}, B, \vec{U_2}$.

4.3 Using System F as source calculus

In addition to being compatible with different evaluation strategies, F_Υ also allows for different type systems in the source calculus. Exploring this question in detail is beyond the scope of this paper, but we just want to illustrate how we could rather have chosen a source calculus typed with System F. Since the second-order quantification in System F is intended to be substituted by any type, through the translations second-order variables should behave as the translation of types. In particular, they should have access to the current store type. The corresponding translations, that have to be defined at the lowest level, are reminiscent of forcing translations where types are involved [15, 25]:

$$\Upsilon \triangleright_{\mathsf{v}} \forall X.F \triangleq \forall (X : \Upsilon \to T).\Upsilon \triangleright_{\mathsf{v}} F \qquad \Upsilon \triangleright_{\mathsf{v}} X \triangleq X\,\Upsilon$$

5 Conclusion

Conclusion Through the definition of F_Υ, we isolated the key ingredients necessary to the definition of well-typed continuation-and-environment-passing style translations:

1. terms to represent and manipulate typed stores,
2. explicit coercions to witness store extensions.

The system we propose has the benefits of being highly parametric: not only is it suitable for defining CEPS translations for simply typed source calculi with global environments evaluated in call-by-need, call-by-name or call-by-value; it is also compatible with different type systems. In addition, while we chose to implement stores with lists, we could as well have chosen to leave their concrete representation abstract. In particular, one could use the very same architecture to handle source calculi where environments are defined through different data structures (tree, records, etc).

About environments and forcing On the logical side, the translation of types amounts to a (Kripke) forcing translation (for the environment-passing part), interleaved with a negative translation (for the continuation-passing part). Actually, the connection between forcing and the environment-passing style translation does not come as a surprise. It is folklore that the local state monad (used to give a meaning to memory states in functional programming) can be categorically interpreted by means of presheaves construction [24, 31]. Interestingly, presheaves also give a semantics to Kripke models or Cohen forcing [15, 23, 27]. Last but not least, the analysis of Cohen forcing in the framework of Krivine realizability [19, 25] relies on an extension of Krivine abstract machine with a cell (containing the forcing condition), which resembles the Krivine realizability interpretation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus in [26]. In short, our typed environment-passing style translation is just another observation of the connection between forcing translations and explicit expandable environments as a side-effect.

Further work This work should open the way to the definition of well-typed compilation transformations for lazily-evaluated calculi. For instance, the MetaCoq project managed to identify a type transformation of programs mapped to the call-by-value evaluation through the erasure procedure [34], but defining a similar transformation for the lazy evaluation will necessarily require to define a typed environment-passing style transformation. On the logical side, as we emphasized in Section 3.3, it seems that our system somehow lies in between Cardelli's system $\mathrm{F}_{<:}$ [7] and dependently typed calculi (to manipulate sized lists). Yet, the precise logical strength of F_Υ is unclear and is still to determine.

Last but not least, we would like to invest whether the $\Box$ can be considered as a modality, and in particular whether we could optimize the translation by using a polarized source calculus, inserting the $\Box$ in the translation only when polarity changes.

Acknowledgments

Both authors would like to thank the anonymous reviewers for their accurate remarks. Moreover, the first author thanks Keiko Nakata, as well as José Carlos Espírito Santo, Luís Pinto, Zena Ariola, Paul Downen, Alexis Saurin and Rossen Mikhov for initial discussions on typing the continuation-passing style semantics of call-by-need λ-calculus. The second author was supported by the Paris Ile-de-France Region and would like to thank Théo Winterhalter and Simon Boulier for their valuable comments on this work and the companion Coq development.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. 1990. Explicit substitutions. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 31–46. https://doi.org/10.1145/96709.96712

[2] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling Abstract Machines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 363–376. https://doi.org/10.1145/2628136.2628154

[3] Beniamino Accattoli and Bruno Barras. 2017. Environments and the Complexity of Abstract Machines. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP '17)*. Association for Computing Machinery, New York, NY, USA, 4–16. https://doi.org/10.1145/3131851.3131855

[4] Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA.

[5] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. 2012. Classical Call-by-Need Sequent Calculi: The Unity of Semantic Artifacts. In *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings (Lecture Notes in Computer Science)*, Tom Schrijvers and Peter Thiemann (Eds.). Springer, New York, NY, USA, 32–46. https://doi.org/10.1007/978-3-642-29822-6

[6] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Proceedings of CPP 2017*. ACM, New York, NY, USA, 182–194. https://doi.org/10.1145/3018610.3018620

[7] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. *An extension of system F with subtyping*. Springer Berlin Heidelberg, Berlin, Heidelberg, 750–770. http://dx.doi.org/10.1007/3-540-54415-1_73

[8] Pierre Crégut. 2007. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation* 20, 3 (01 Sep 2007), 209–230. https://doi.org/10.1007/s10990-007-9015-z

[9] Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of ICFP 2000 (SIGPLAN Notices 35(9))*. ACM, New York, NY, USA, 233–243. https://doi.org/10.1145/351240.351262

[10] Pierre-Évariste Dagand, Lionel Rieg, and Gabriel Scherer. 2019. Dependent Pearl: Normalization by realizability. (2019). arXiv:cs.PL/1908.09123

[11] Nicolaas G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381 – 392. https://doi.org/10.1016/1385-7258(72)90034-0

[12] Matthias Felleisen and Daniel P. Friedman. 1986. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*. North-Holland, New York, NY, USA, 193–217.

[13] Matthias Felleisen and Amir Sabry. 1999. Continuations in programming practice: Introduction and survey. (1999). https://www.cs.indiana.edu/~sabry/papers/continuations.ps Manuscript.

[14] Timothy G. Griffin. 1990. A Formulae-as-type Notion of Control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/96709.96714

[15] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. 2016. The Definitional Side of the Forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NY, USA, 367–376. https://doi.org/10.1145/2933575.2935320

[16] Saul A. Kripke. 1963. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica* 16, 1963 (1963), 83–94.

[17] Jean-Louis Krivine. 1993. *Lambda-calculus, types and models*. Masson.

[18] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. In *Higher Order and Symbolic Computation*, Vol. 20. Springer, New York, NY, USA, 199–207. https://doi.org/10.1007/s10990-007-9018-9

[19] Jean-Louis Krivine. 2011. Realizability algebras: a program to well order R. *Logical Methods in Computer Science* 7, 3 (2011), 1–47.

[20] Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

[21] Frédéric Lang. 2007. Explaining the lazy Krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation* 20, 3 (01 Sep 2007), 257–270. https://doi.org/10.1007/s10990-007-9013-1

[22] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language.* Technical report 117. INRIA.

[23] Saunders MacLane and Ieke Moerdijk. 1992. *Sheaves in Geometry and Logic.* Springer, New York, NY, USA. https://doi.org/10.1007/978-1-4612-0927-0

[24] Paul-André Melliès. 2014. *Local States in String Diagrams.* Springer International Publishing, Cham, 334–348.

[25] Alexandre Miquel. 2011. Forcing as a Program Transformation. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. IEEE Computer Society, USA, 197–206. https://doi.org/10.1109/LICS.2011.47

[26] Étienne Miquey and Hugo Herbelin. 2018. Realizability Interpretation and Normalization of Typed Call-by-Need λ-calculus with Control. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 276–292. https://doi.org/10.1007/978-3-319-89366-2_15

[27] Ieke Moerdijk and Jaap van Oosten. 2007. Topos Theory. (2007). http://www.staff.science.uu.nl/~ooste110/syllabi/toposmoeder.pdf

[28] Chetan Murthy. 1990. *Extracting constructive content from classical proofs.* Ph.D. Thesis. Cornell University.

[29] Chris Okasaki, Peter Lee, and David Tarditi. 1994. Call-by-Need and Continuation-Passing Style. *Lisp and Symbolic Computation* 7, 1 (1994), 57–82. https://doi.org/10.1007/BF01019945

[30] Michel Parigot. 1997. Proofs of Strong Normalisation for Second Order Classical Natural Deduction. *J. Symb. Log.* 62, 4 (1997), 1461–1479.

[31] Gordon Plotkin and John Power. 2002. *Notions of Computation Determine Monads.* Springer Berlin Heidelberg, Berlin, Heidelberg, 342–356.

[32] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[33] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *J. Funct. Program.* 7, 3 (May 1997), 231–264. https://doi.org/10.1017/S0956796897002712

[34] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

[35] Gerald J. Sussman and Guy L. Steele, Jr. 1975. *An Interpreter for Extended Lambda Calculus.* Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA.

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

a. Milner Abstract Machine

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

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

Figure 11. 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. 11). We first define the following compilation function from states of the MAM to closures of the $\bar{\lambda}\mu\tilde{\mu}$-calculus:

$$[\![t \star \pi \star \tau]\!] \triangleq \langle [\![t]\!]_{\mathrm{t}} \parallel [\![\pi]\!]_{\pi}\rangle [\![\tau]\!]_{\tau}$$

with:

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

where κ 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 α-conversion). Formally, to avoid considering the contexts stored in the environment, we first define a substitution function for co-variables. We write $c\{\tau\}$ for the closure $c\tau'$ in which all the co-variables α bound in τ are recursively substituted by the terms to which they are bound and where τ' consists of the fragment of τ binding only variables:

$$c\{\varepsilon\} \triangleq c \qquad c\{\tau[x := t]\} \triangleq (c\{\tau\})[x := t] \qquad c\{\tau[\alpha := E]\} \triangleq (c[E/\alpha])\{\tau\}$$

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 $c\{\tau\} = c'\{\tau'\}$.

Proposition A.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 now need to take the dump into account. As we explain in Section 1.4, 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:

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

with:

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

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

Once again, it is straightforward to check that:

Proposition A.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. □

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

$$
\begin{array}{lll}
\textbf{Forcing contexts} & F & ::= t \cdot E \mid \boldsymbol{\kappa} \\
\textbf{Catchable contexts} & E & ::= F \mid \alpha \mid \tilde{\mu}[x].\langle x \parallel F \rangle\tau \\
\textbf{Evaluation contexts} & e & ::= E \mid \tilde{\mu}x.c
\end{array}
$$

$$
\begin{array}{lcl}
(\textsc{Let}) & \langle t \parallel \tilde{\mu}x.c \rangle\tau & \to \quad c\tau[x := t] \\
(\textsc{Catch}) & \langle \mu\alpha.c \parallel E \rangle\tau & \to \quad c\tau[\alpha := E] \\
(\textsc{Lookup}_\alpha) & \langle V \parallel \alpha \rangle\tau[\alpha := E]\tau' & \to \quad \langle V \parallel E \rangle\tau[\alpha := E]\tau' \\
(\textsc{Lookup}_x) & \langle x \parallel F \rangle\tau[x := t]\tau' & \to \quad \left\langle t \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F \rangle\tau' \right\rangle\tau \\
(\textsc{Restore}) & \left\langle V \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F \rangle\tau' \right\rangle\tau & \to \quad \langle V \parallel F \rangle\tau[x := V]\tau' \\
(\textsc{Beta}) & \langle \lambda x.t \parallel u \cdot E \rangle\tau & \to \quad \left\langle u \,\middle\|\, \tilde{\mu}x.\langle t \parallel E \rangle \right\rangle\tau
\end{array}
$$

Figure 12. The $\overline{\lambda}_{[lv\tau\star]}$-calculus

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

B.1 Definitions

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

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

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

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

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

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

Figure 13. Typing rules for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

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. 12, while the type system, defined in [26], is given in Fig. 13. Finally, Ariola *et al.*'s original untyped CPS translation is given in Fig. 14.

B.2 The necessity of α-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 α-conversion on-the-fly. Consider for instance a command formed by a term of the shape $t = \mu\alpha.\left\langle u \,\middle\|\, \tilde{\mu}x.\langle x \parallel \alpha \rangle \right\rangle$ 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 [26], however, reducing this command (without α-conversion) would loop forever because of the

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

Figure 14. Ariola *et al.* untyped CPS translation

auto-reference $[x := x]$ in the environment:

$$
\begin{array}{rl}
\left\langle \mu\alpha.\left\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\right\rangle \middle\| \tilde{\mu}x.\langle x \parallel F\rangle \right\rangle & \to \langle x \parallel F\rangle[x := \mu\alpha.\left\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\right\rangle] \\
& \to \left\langle \mu\alpha.\left\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\right\rangle \middle\| \tilde{\mu}[x].\langle x \parallel F\rangle \right\rangle \\
& \to \left\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\right\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle] \\
& \to \langle x \parallel \alpha\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle, x := u] \\
& \to \left\langle x \middle\| \tilde{\mu}[x].\langle x \parallel F\rangle\right\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle, x := u] \\
& \to \langle x \parallel F\rangle[\alpha := \tilde{\mu}[x].\langle x \parallel F\rangle, x := u, x := x] \to \ldots
\end{array}
$$

While a simple α-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 14 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 α-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:

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

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 α-conversion to prevent the formation of the cyclic reference $[x := [\![x]\!]_V]$. This is why we need either to be able to perform α-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 $\overline{\lambda}_{[l\upsilon\tau\star]}$, using de Bruijn levels. As we observed in the previous example, the issue arises when adding a binding $[x := \ldots]$ 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 λ 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:

$$\begin{aligned}
&\left\langle \mu\alpha.\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle \,\middle\|\, \tilde{\mu}x.\langle x \parallel F\rangle \right\rangle \to \langle x_0 \parallel F\rangle[^0\mu\alpha.\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle] \\
\to\ &\left\langle \mu\alpha.\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle \right\rangle \to \langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha_0\rangle\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle] \\
&\to \langle x_1 \parallel \alpha_0\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle, {}^1u] \to \left\langle x_1 \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle\right\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle, {}^1u] \\
&\to \langle x_1 \parallel F\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle, {}^1u, {}^2x_1] \to \left\langle u \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle[^2x_1]\right\rangle[^0\tilde{\mu}[x].\langle x \parallel F\rangle]
\end{aligned}$$

where the exponents $^0, {}^1, \ldots$ 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 $[\![-]\!]^\sigma_t$ where σ is a substitution used to rename variables. To compact the notations, we write $[^x_m|^\alpha_\gamma|\ldots]$ for the renaming substitution $[x := m, \alpha := \gamma, \ldots]$, 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]\!]^{[^{x}_{m}]}_{c}\ [m := [\![t]\!]_{t}] \\
&= [\![x]\!]^{[^{x}_{m}]}_{t}\ [m := [\![t]\!]^{\varepsilon}_{t}]\ [\![F]\!]^{[^{x}_{m}]}_{F} \\
&= [\![\mu\alpha.\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle]\!]^{[^{x}_{m}]}_{t}\ \varepsilon\ (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}) \\
&= [\![\langle u \parallel \tilde{\mu}x.\langle x \parallel \alpha\rangle\rangle]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}] \\
&= [\![\tilde{\mu}x.\langle x \parallel \alpha\rangle]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{e}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}]\ [\![u]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t} \\
&= [\![\langle x \parallel \alpha\rangle]\!]^{[x:=m,\alpha:=\gamma,x:=n]}_{c}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}, n := [\![u]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t}] \\
&= [\![\alpha]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}|^{x}_{n}]}_{E}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}, n := [\![u]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t}][\![x]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}|^{x}_{n}]}_{V} \\
&= (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V])\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}, n := [\![u]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t}]\ [\![x]\!]^{[^{\alpha}_{\gamma}|^{x}_{n}]}_{V} \\
&\to [\![x]\!]^{[^{\alpha}_{\gamma}|^{x}_{n}]}_{V}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}, n := [\![u]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t}, m := [\![x]\!]^{[^{\alpha}_{\gamma}|^{x}_{n}]}_{t}] \\
&= [\![x]\!]^{[^{\alpha}_{\gamma}|^{x}_{n}]}_{V}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^{t} V]\ [\![F]\!]^{[^{x}_{m}]}_{F}, n := [\![u]\!]^{[^{x}_{m}|^{\alpha}_{\gamma}]}_{t}, m := [\![x]\!]^{[^{\alpha}_{\gamma}|^{x}_{n}]}_{t}]
\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 $j \geq i$ (resp. α_j) have been replaced by x_{j+n} (resp. α_{j+n}). Given i, n two natural numbers, we define:

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

C Context-free abstract machines

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

$$
\begin{array}{rcl}
\langle t \parallel \tilde{\mu}x.c\rangle_{\mathsf{e}}\tau & \to & c_{\mathsf{e}}\tau[x := t] \\
\langle t \parallel E\rangle_{\mathsf{e}}\tau & \to & \langle t \parallel E\rangle_{\mathsf{t}}\tau \\
\langle \mu\alpha.c \parallel E\rangle_{\mathsf{t}}\tau & \to & c_{\mathsf{e}}\tau[\alpha := E] \\
\langle V \parallel E\rangle_{\mathsf{t}}\tau & \to & \langle V \parallel E\rangle_{\mathsf{E}}\,\tau \\
\langle V \parallel \alpha\rangle_{\mathsf{E}}\,\tau[\alpha := E]\tau' & \to & \langle V \parallel E\rangle_{\mathsf{E}}\,\tau[\alpha := E]\tau' \\
\left\langle V \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle\tau'\right\rangle_{\mathsf{E}}\,\tau & \to & \langle V \parallel F\rangle_{\mathsf{V}}\,\tau[x := V]\tau' \\
\langle V \parallel F\rangle_{\mathsf{E}}\,\tau & \to & \langle V \parallel F\rangle_{\mathsf{V}}\,\tau \\
\langle x \parallel F\rangle_{\mathsf{V}}\,\tau[x := t]\tau' & \to & \left\langle t \,\middle\|\, \tilde{\mu}[x].\langle x \parallel F\rangle\tau'\right\rangle_{\mathsf{t}}\tau \\
\langle v \parallel E\rangle_{\mathsf{V}}\,\tau & \to & \langle v \parallel F\rangle_{\mathsf{V}}\,\tau \\
\langle v \parallel u \cdot E\rangle_{\mathsf{F}}\,\tau & \to & \langle v \parallel u \cdot E\rangle_{\mathsf{v}}\tau \\
\langle \lambda x.t \parallel u \cdot E\rangle_{\mathsf{v}}\tau & \to & \left\langle u \,\middle\|\, \tilde{\mu}x.\langle t \parallel E\rangle\right\rangle_{\mathsf{e}}\tau
\end{array}
$$

Figure 15. 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

$$
\begin{array}{rcll}
\langle t \parallel \tilde{\mu}x_i.c\rangle_{\mathsf{e}}\tau & \to & c[x_n/x_i]_{\mathsf{e}}\tau[x_n := t] & \text{with } |\tau| = n \\
\langle t \parallel E\rangle_{\mathsf{e}}\tau & \to & \langle t \parallel E\rangle_{\mathsf{t}}\tau & \\
\langle \mu\alpha_i.c \parallel E\rangle_{\mathsf{t}}\tau & \to & c[\alpha.n/\alpha_i]_{\mathsf{e}}\tau[\alpha_n := E] & \text{with } |\tau| = n \\
\langle x_n \parallel E\rangle_{\mathsf{t}}\tau & \to & \langle \tau(n) \parallel E\rangle_{\mathsf{t}}\tau & \\
\langle V \parallel E\rangle_{\mathsf{t}}\tau & \to & \langle V \parallel E\rangle_{\mathsf{E}}\,\tau & \\
\langle V \parallel \alpha_n\rangle_{\mathsf{E}}\,\tau & \to & \langle V \parallel \tau(n)\rangle_{\mathsf{E}}\,\tau & \\
\langle V \parallel u \cdot E\rangle_{\mathsf{E}}\,\tau & \to & \langle V \parallel u \cdot E\rangle_{\mathsf{V}}\,\tau & \\
\langle \lambda x.t \parallel u \cdot E\rangle_{\mathsf{v}}\tau & \to & \left\langle u \,\middle\|\, \tilde{\mu}x.\langle t \parallel E\rangle\right\rangle_{\mathsf{e}}\tau &
\end{array}
$$

Figure 16. Context-free abstract machine for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with environments

D Properties of System F_Υ

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

D.1 Coercions

Lemma D.1. *If σ_1 is in normal form, then the following rule is admissible:*

$$\frac{\Gamma \vdash \sigma_0 : \Upsilon_0' <: \Upsilon_0 \quad \Gamma \vdash \sigma_1 : \Upsilon_1' <: \Upsilon_1}{\Gamma \vdash \sigma_0; \sigma_1 : \Upsilon_0'; \Upsilon_1' <: \Upsilon_0; \Upsilon_1}$$

Remember that 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 $$\sigma ::= \varepsilon \mid \sigma^+ \mid \Uparrow \sigma$$

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

$$\sigma_1^+ \odot \sigma_0^+ \triangleq (\sigma_1 \odot \sigma_0)^+ \qquad \Uparrow \sigma_1 \odot \sigma_0 \triangleq \Uparrow (\sigma_1 \odot \sigma_0)$$
$$\sigma_1^+ \odot \Uparrow \sigma_0 \triangleq \Uparrow (\sigma_1 \odot \sigma_0) \qquad \varepsilon \odot \sigma_0 = \sigma_1^+ \odot \varepsilon \triangleq \varepsilon$$

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

Lemma D.2 (Composition of normal forms)**.** *If σ, σ' 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 σ'. □

This 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[17]. We then 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 D.3 (Computing normal forms)**.** *For any computable σ, if $\Gamma \vdash \sigma : \Upsilon' <: \Upsilon$ then there exists σ_n in normal form such that $\Gamma \vdash \sigma_n : \Upsilon' <: \Upsilon$.*

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

It is worth noting that for any $\sigma, \Upsilon, \Upsilon'$, if $\vdash \sigma : \Upsilon' <: \Upsilon$, then σ is necessarily computable.

Corollary D.4. *If $\vdash \sigma : \Upsilon' <: \Upsilon$, then $|\Upsilon| \leq |\Upsilon'|$.*

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

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 Υ to its corresponding index in Υ'.

Formally, if σ 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:

$$\mathrm{dom}(\varepsilon) \triangleq 0 \quad \mathtt{codom}(\varepsilon) \triangleq 0 \qquad \mathrm{dom}(\sigma^+) \triangleq \mathrm{dom}(\sigma) + 1 \quad \mathtt{codom}(\sigma^+) \triangleq \mathtt{codom}(\sigma) + 1 \qquad \mathrm{dom}(\Uparrow \sigma) \triangleq \mathrm{dom}(\sigma) \quad \mathtt{codom}(\Uparrow \sigma) \triangleq \mathtt{codom}(\sigma) + 1$$

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

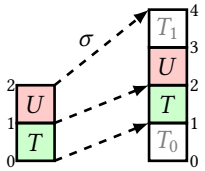
$$[\![\varepsilon]\!] \triangleq \{0 \mapsto 0\} \qquad [\![\sigma^+]\!] \triangleq [\![\sigma]\!] \cup \{\mathrm{dom}(\sigma) \mapsto \mathtt{codom}(\sigma)\} \qquad [\![\Uparrow \sigma]\!] \triangleq \begin{cases} n < \mathrm{dom}(\sigma) \mapsto [\![\sigma]\!](n) \\ n = \mathrm{dom}(\sigma) \mapsto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

Notice that $[\![\sigma]\!]$ is always a strictly monotone function.

[17] We could also include $\Uparrow_\Upsilon \sigma_n$ and $\sigma_n^{+\Upsilon}$ where Υ is in normal form, but for the sake of simplicity, let us just focus on this simpler fragment.

Example D.5. As an example, we let the reader verify that for any types T, U, T_0, T_1, if we denote by σ for the coercion $\Uparrow((\Uparrow\varepsilon)^{++})$ which is in normal form, we have:

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



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

The previous definitions are indeed in adequacy with the intuition we gave above:

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

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

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 $\mathsf{dom}(\sigma)$. □

The previous proposition opens the way for proving properties of computable coercions through their associated functions. For instance, we have (remember that we define $\mathsf{id}_\Upsilon = \varepsilon^{+_\Upsilon}$):

Proposition D.7 (Partial order)**.** *In the empty context, the subtyping relation $<:$ is an order relation on store types.*

1. *For any Υ, we have* $\dfrac{}{\vdash \mathsf{id}_\Upsilon : \Upsilon <: \Upsilon}\ (<:_{\mathsf{id}})$ *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 D.6, since $[\![\sigma]\!]$ and $[\![\sigma']\!]$ are two strictly monotone functions from $[0, |\Upsilon|]$ to itself (by Corollary D.4 we have $|\Upsilon'| = |\Upsilon|$), they are necessarily the identity. Equivalently, we could have seen that necessarily σ is of the shape σ_0^+ (and so is σ'), so that $\Upsilon' = \Upsilon'_0, T$ and $\Upsilon = \Upsilon_0, T$, from which we can conclude by an easy induction. □

D.2 Subject reduction

First, it is clear that the type system is compatible with a weakening rule:

Lemma D.8 (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 Γ' and not in Γ. □

Before proving subject reduction, we need to show 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 σ, by precomposing terms with σ. Remember that given a coercion σ of type $\Upsilon' <: \Upsilon$ and a term t whose type is of the shape $\forall Y <: \Upsilon.A$, we defined:

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

Lemma D.9. *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:

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

where we use Lemma D.8 to weaken $\Gamma, \sigma : X <: \Upsilon_1$. □

As a Corollary, since we require ($\mathrm{Eq}_{\blacktriangleright}$) that $\Upsilon \blacktriangleright F$ is always of the shape $\forall Y <: \Upsilon.\mathcal{F}(Y, F)$, we get that

Corollary D.10. *The following rule is admissible:*

$$\frac{\Gamma \vdash t : \Upsilon_0 \blacktriangleright F \quad \Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash (\uparrow^{\sigma} t) : \Upsilon_1 \blacktriangleright F}$$

Lemma D.11. *If σ_1 is in normal form, then the following rule is admissible:*

$$\frac{\Gamma \vdash \sigma_0 : \Upsilon_0' <: \Upsilon_0 \quad \Gamma \vdash \sigma_1 : \Upsilon_1' <: \Upsilon_1}{\Gamma \vdash \sigma_0; \sigma_1 : \Upsilon_0'; \Upsilon_1' <: \Upsilon_0; \Upsilon_1}$$

We can now verify type safety with respect to reductions:

Theorem 3.4 (Subject reduction)**.** *For any context Γ, any type T and any terms t, t', if $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.*

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 $\Gamma, \delta : \Upsilon_0 \triangleright_\tau \Upsilon_1, \Gamma' \vdash t : B$ and $\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon_1$, 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 $\Gamma, \Gamma'[\Upsilon'/Y] \vdash t[\sigma/s] : B[\Upsilon'/Y]$.
4. If $\Gamma \vdash t : B$ then $\Gamma[\Upsilon/Y] \vdash t[\Upsilon/Y] : B[\Upsilon/Y]$.

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. The (almost) only interesting case is the one induced by the following reduction rule:

$$\uparrow^{\sigma}_{\Upsilon,F} \tau[t] \to_\tau (\uparrow^{\sigma}_{\Upsilon} \tau)[\uparrow^{\sigma^{+\Upsilon}} t]$$

Indeed, if we have :

$$\dfrac{\Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0 \qquad \dfrac{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \quad \Gamma \vdash t : \Upsilon_0; \Upsilon \blacktriangleright F}{\Gamma \vdash \tau[t] : \Upsilon_0 \triangleright_\tau \Upsilon, F}\,([t])}{\Gamma \vdash \uparrow^{\sigma}_{\Upsilon,F} \tau[t] : \Upsilon_1 \triangleright_\tau \Upsilon, F}\,(\uparrow\tau)$$

we can derive (using Corollary D.10):

$$\dfrac{\dfrac{\Gamma \vdash \sigma : \Upsilon_1 <: \Upsilon_0 \quad \Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon}{\Gamma \vdash \uparrow^{\sigma}_{\Upsilon} \tau : \Upsilon_0 \triangleright_\tau \Upsilon}\,(\uparrow\tau) \qquad \dfrac{\Gamma \vdash \sigma^{+\Upsilon} : \Upsilon_1; \Upsilon <: \Upsilon_0; \Upsilon \quad \Gamma \vdash t : \Upsilon_0; \Upsilon \blacktriangleright F}{\Gamma \vdash \uparrow^{\sigma^{+\Upsilon}} t : \Upsilon_1; \Upsilon \blacktriangleright F}}{\Gamma \vdash (\uparrow^{\sigma}_{\Upsilon} \tau)[\uparrow^{\sigma^{+\Upsilon}} t] : \Upsilon_1 \triangleright_\tau \Upsilon, F}\,(\uparrow\tau)$$

□

Remark 2 (About the (split) -rule)**.** *We shall attract the reader's attention on a specificity of the* (split) *that she may have observed. Remember that we defined the rule by :*

$$\dfrac{\begin{array}{llll}\Gamma \vdash \tau : \Upsilon' & \Gamma \vdash \sigma : \Upsilon' <: \Upsilon & \Upsilon = [\![\Gamma_0]\!], F, [\![\Gamma_1]\!] & |\Gamma_0| = n \\ \Gamma', s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \blacktriangleright F, s_1 : (Y_0, F; Y_1 <: \Upsilon), \delta_1 : (Y_0, F) \triangleright_\tau Y_1 \vdash t : A' & & & \end{array}}{\Gamma \vdash \mathtt{split}\ \tau\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon' <: \Upsilon\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ t : A'}\,(\mathtt{split})$$

where $\Gamma' = \Gamma[Y_0, F; Y_1/\Upsilon']$ and $A' = A[Y_0, F; Y_1/\Upsilon']$ (which is an abuse of notations to say that $\Gamma' = \Gamma_0[\Upsilon'/\bullet], \Gamma = \Gamma_0[Y_0, F; Y_1/\bullet]$ for some Γ_0, etc). The latter equalities are necessary for identifying Υ' and its cutting in the typing derivation for t, which may contain at the same time terms build on $s_0, s_1, \ldots$ thus refering to Y_0, Y_1 and terms refering directly to Υ'. To avoid more syntax, we chose to take this equality into account directly with a substitution, but we also could have simply considered two extra coercion variables $s_i : Y_0, F; Y_1 <: \Upsilon'$ and $s_j : \Upsilon' <: Y_0, F; Y_1$ in the premise. In addition to giving access to subterm in t to both representations of Υ', it also enforces the equality between Υ' and $Y_0, F; Y_1$ (remember that $<:$ induces an order relation, Proposition D.7). In particular, the reduction rule for split *should then instantiate those two coercions with $\mathrm{id}_{\Upsilon'}$. This approach would result in the following typing rule:*

$$\dfrac{\begin{array}{llll}\Gamma \vdash \tau : \Upsilon' & \Gamma \vdash \sigma : \Upsilon' <: \Upsilon & \Upsilon = [\![\Gamma_0]\!], F, [\![\Gamma_1]\!] & |\Gamma_0| = n \\ \Gamma, s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \blacktriangleright F, s_1 : (Y_0, F; Y_1 <: \Upsilon), \delta_1 : (Y_0, F) \triangleright_\tau Y_1, s_i : (Y_0, F; Y_1) <: \Upsilon', s_j : \Upsilon' <: (Y_0, F; Y_1) \vdash t : A & & & \end{array}}{\Gamma \vdash \mathtt{split}\ \tau\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon' <: \Upsilon\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1), s_i, s_j\ \mathtt{in}\ t : A'}\,(\mathtt{split})$$

together with the following reduction rule:

$$\texttt{split}\ \tau_0[u];\tau_1\ \texttt{at}\ n\ \texttt{along}\ \sigma : \Upsilon_0, A, \Upsilon_1 <: [\![\Gamma_0]\!], A, [\![\Gamma_1]\!]\ \texttt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1), s_i, s_j\ \texttt{in}\ t$$
$$\rightarrow$$
$$t[\Upsilon_0/Y_0][\sigma_0/s_0][\tau_0/\delta_0][u/x][\Upsilon_1/Y_1][\tau_1/\delta_1][\sigma_1/s_1][\mathsf{id}_{\Upsilon_0, A\,\Upsilon_1}/s_i][\mathsf{id}_{\Upsilon_0, A\,\Upsilon_1}/s_j]$$

where $n = |\Gamma_0|$, $|\tau_0| = |\Upsilon_0|$ *and* $\Upsilon_0, \Upsilon_1, \sigma_0, \sigma_1$ *are again as in Lemma 3.2. This reduction is also safe with respect to typing.*

D.3 Normalization

The proof of normalization for F_Υ that we present in this section is inspired from techniques of Krivine's classical realizability [18], whose notations we borrow. Actually, it is also very close to a proof by reducibility[18]. In a nutshell, to each type A is associated a set $|A|$ of terms whose execution is guided by the structure of A. These terms are the ones usually called *realizers* in Krivine's classical realizability. Their definition is in fact indirect, and is done by orthogonality to a set of "correct" computations, called a *pole*. The choice of this set is central when studying models induced by classical realizability for second-order-logic, but in the present case we only pay attention to the particular pole of terminating computations. This is where lies one of the difference with usual proofs by reducibility, where everything is done with respect to SN, while our definition are parametric in the pole (which is chosen to be SN in the end). The adequacy lemma, which is the central piece, consists in proving that typed terms belong to the corresponding sets of realizers, and are thus normalizing.

We try to remain as concise as possible, for a comprehensive introduction to Krivine realizability and normalization proof in that context, we refer the reader to Dagand, Rieg and Scherer's [10].

Abstract machine We begin by adapting the Krivine Abstract Machine to our calculus:

$$\begin{array}{lrcl}
\textbf{Terms} & t & ::= & \boldsymbol{k} \mid x \mid \lambda Y.t \mid t\,\Upsilon \mid \lambda s.t \mid t\,\sigma \mid \lambda\delta.t \mid t\,\tau \mid \lambda x.t \mid t\,u \\
\textbf{Stacks} & \pi & ::= & \bullet \mid \Upsilon \cdot \pi \mid \sigma \cdot \pi \mid \tau \cdot \pi \mid t \cdot \pi \\
\textbf{Processus} & p & ::= & t \star \pi
\end{array}$$

We then define the reduction rules $p \succ p'$. We first define the usual rules for abstractions/applications (where $M \in \Upsilon \mid \sigma \mid \tau \mid u$):

$$\begin{array}{rcl}
t\,M \star \pi & \succ & t \star M \cdot \pi \\
\lambda Y.t \star \Upsilon \cdot \pi & \succ & t[\Upsilon/Y] \star \pi \\
\lambda s.t \star \sigma \cdot \pi & \succ & t[\sigma/Y] \star \pi \\
\lambda\delta.t \star \tau \cdot \pi & \succ & t[\tau/Y] \star \pi \\
\lambda x.t \star u \cdot \pi & \succ & t[u/Y] \star \pi
\end{array}$$

to which we add a rule for splitting stores:

$$\texttt{split}\ \tau_0[u]\tau_1\ \texttt{at}\ n\ \texttt{along}\ \sigma : \Upsilon_0, A, \Upsilon_1 <: [\![\Gamma_0]\!], A, [\![\Gamma_1]\!]\ \texttt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \texttt{in}\ t \star \pi$$
$$\succ$$
$$t[\Upsilon_0/Y_0][\sigma_0/s_0][\tau_0/\delta_0][u/x][\Upsilon_1/Y_1][\tau_1/\delta_1][\sigma_1/s_1] \star \pi$$

where $n = |\Gamma_0|$ and $\Upsilon_0, \Upsilon_1, \sigma_0, \sigma_1$ are as in Lemma 3.2 (i.e. $|\Upsilon_0| = [\![\sigma]\!](n)$, $\sigma_0 : \Upsilon_0 <: \Gamma_0$, etc.), and $|\tau_0| = |\Upsilon_0|$.

Finally, we add two contextual rules for reducing coercions and stores in `split`:

$$\begin{array}{rcll}
\texttt{split}\ \tau\ \texttt{at}\ n\ \texttt{along}\ \ldots\ \texttt{as}\ \ldots\ \texttt{in}\ t \star \pi & \succ & \texttt{split}\ \tau'\ \texttt{at}\ n\ \texttt{along}\ \ldots\ \texttt{as}\ \ldots\ \texttt{in}\ t \star \pi & (\text{if } \tau \rightarrow_\tau \tau') \\
\texttt{split}\ \tau_n\ \texttt{at}\ n\ \texttt{along}\ \sigma : \ldots\ \texttt{as}\ \ldots\ \texttt{in}\ t \star \pi & \succ & \texttt{split}\ \tau\ \texttt{at}\ n\ \texttt{along}\ \sigma' : \ldots\ldots\ \texttt{as}\ \ldots\ \texttt{in}\ t \star \pi & (\text{if } \sigma \rightarrow_\sigma \sigma')
\end{array}$$

where τ_n is in normal form (in other words, we first reduce stores then coercions).

Realizability interpretation We now focus on the definition of the Krivine realizability interpretation, which relies on the definition truth and falsity values (which are themselves defined with respect to a pole).

Definition D.12 (Pole)**.** A subset $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$ is said to be *closed under anti-reduction* whenever for all closed $p, p' \in \Lambda \star \Pi$, if $p' \in \perp\!\!\!\perp$ and $p \succ p'$ then $p \in \perp\!\!\!\perp$. A *pole* is defined as any set of closed processes that is closed under anti-reduction.

Definition D.13 (Valuation)**.** A *substitution*, written ρ, is a function mapping term variables to closed terms (written $\rho, x := t$), coercion variables to closed coercions (written $\rho, s := \sigma$), store variables to closed stores (written $\rho, \delta := \tau$) and store type variable to closed store types (written $\rho, Y := \Upsilon$).

We write $\rho(t)$ (resp. $\rho(A)$, $\rho(\Upsilon)$, etc.) the term t where the substitution ρ has been performed.

[18] See for instance the proof of normalization for system D presented in [17, 3.2].

Given a fixed pole $\perp\!\!\!\perp$, we define the interpretation of closed types as follows:

$$\begin{array}{rcl} \|X\| & = & \{\bullet\} \\ \|\forall Y.T\| & = & \bigcup_{\Upsilon\in\mathcal{F}^*}\{\Upsilon\cdot\pi : \pi\in\|T[\Upsilon/Y]\|\} \\ \|\Upsilon' <: \Upsilon \to T\| & = & \{\sigma\cdot\pi : \sigma \Vdash \Upsilon' <: \Upsilon \wedge \pi\in\|T\|\} \\ \|\Upsilon \triangleright_\tau \Upsilon' \to T\| & = & \{\tau\cdot\pi : \tau \Vdash \Upsilon' \triangleright_\tau \Upsilon \wedge \pi\in\|T\|\} \\ \|U \to T\| & = & \{u\cdot\pi : u\in|U| \wedge \pi\in\|T\|\} \\ |T| & = & \|T\|^{\perp\!\!\!\perp} = \{t : \forall\pi\in\|T\|, t\star\pi\in\perp\!\!\!\perp\} \end{array}$$

where $\mathcal{F}^*$ is defined as the set of closed store types (*i.e.* the set of lists of source types corresponding to the subsyntax $\Upsilon_0 = \emptyset \mid \Upsilon_0, F$).

We say that (where all terms and types are closed):

1. $\sigma \Vdash \Upsilon' <: \Upsilon$ if $\vdash \sigma : \Upsilon' <: \Upsilon$
2. $\tau \Vdash \Upsilon' \triangleright_\tau \Upsilon$ if $\forall \tau', \tau' \Vdash \Upsilon' \Rightarrow \tau'; \tau \Vdash \Upsilon'; \Upsilon$
3. $[] \Vdash \emptyset$
4. $\tau[t] \Vdash \Upsilon, F$ if $\tau \Vdash \Upsilon$ and $t \in |\Upsilon \blacktriangleright F|$

We close these relations by anti-reduction with respect to $\to_\sigma$ and $\to_\tau$, that is:

$$\sigma \to_\sigma \sigma' \ \wedge\ \sigma' \Vdash \Upsilon' <: \Upsilon \quad\Rightarrow\quad \sigma \Vdash \Upsilon' <: \Upsilon \qquad\text{and}\qquad \tau \to_\sigma \tau' \ \wedge\ \tau' \Vdash \Upsilon \quad\Rightarrow\quad \tau \Vdash \Upsilon$$

Given a closed context Γ, we say that a substitution ρ *realizes* Γ, which we write $\rho \Vdash \Gamma$, if:

1. for any $Y \in \Gamma$, $\rho(Y)$ is defined
2. for any $(s : \Upsilon' <: \Upsilon) \in \Gamma$, we have $\rho(s) \Vdash \rho(\Upsilon') <: \rho(\Upsilon)$
3. for any $(\delta : \Upsilon' \triangleright_\tau \Upsilon) \in \Gamma$, we have $\rho(\delta) \Vdash \rho(\Upsilon') \triangleright_\tau \rho(\Upsilon)$
4. for any $(x : A) \in \Gamma$, we have $\rho(x) \Vdash \rho(A)$

Definition D.14 (Adequacy)**.** We say that a typing judgment $\Gamma \vdash t : A$ (resp. $\Gamma \vdash \sigma : \Upsilon' <: \Upsilon$, $\Gamma \vdash \tau : \Upsilon' \triangleright_\tau \Upsilon$) is valid with respect to the interpretation if for any substitution ρ such that $\rho \Vdash \Gamma$, then $\rho(t) \Vdash \rho(A)$ (resp. $\rho(s) \Vdash \rho(\Upsilon') <: \rho(\Upsilon)$, $\rho(\delta) \Vdash \rho(\Upsilon') \triangleright_\tau \rho(\Upsilon)$). We say that a typing rule $\dfrac{J_1 \quad \dots \quad J_n}{J}$ is adequate if the validity of the premises $J_1, \dots, J_n$ entails the validity of the conclusion J.

Proposition D.15 (Adequacy)**.** *All the typing rules (except the rule* (*c*) *for constants) are adequate.*

Proof. By case analysis. In each case we assume a substitution $\rho \Vdash \Gamma$ and to ease readibility, we write t_ρ (resp. A_ρ) for $\rho(t)$ (resp. $\rho(A)$).

• **Case** $\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\,(\mathrm{Ax})$. By definition, since $\rho \vdash \Gamma$ and $(x : A) \in \Gamma$, we have that $\rho(x) \Vdash A_\rho$.

• **Case** $\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\lambda)$. Let $u\cdot\pi \in \|A \to B\|$, *i.e.* $u \Vdash A$ and $\pi \in \|B\|$. Then $\lambda x.t_\rho \star u\cdot\pi \succ t_\rho[u/x] \star \pi \in \perp\!\!\!\perp$ by induction hypothesis, since $t_\rho[u/x] = t_{\rho[x:=u]}$ and $\rho[x := u] \Vdash \Gamma, x : A$. We conclude by anti-reduction.

• **Case** $\dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\,(@)$. Let $\pi \in \|B\|$, by induction hypothesis we have $t_\rho \Vdash A \to B$ and $u_\rho \Vdash A$. Therefore, $t_\rho\, u_\rho \star \pi \succ t_\rho \star u_\rho\cdot\pi \in \perp\!\!\!\perp$ since $u_\rho\cdot\pi \in \|A \to B\|$. We conclude by anti-reduction.

• **Case** $\dfrac{}{\Gamma \vdash [] : \emptyset \triangleright_\tau \emptyset}\,(\emptyset)$. Trivial: for all $\tau \Vdash \emptyset$, we have that $\tau; [] = \tau$ and thus $\tau; [] \Vdash \emptyset$.

• **Case** $\dfrac{\Gamma \vdash t : \Upsilon_0 \blacktriangleright F}{\Gamma \vdash [t] : \Upsilon_0 \triangleright_\tau T}\,([t])$. Let $\tau_0 \Vdash \Upsilon_0$. By induction hypothesis, we have that $t_\rho \Vdash \Upsilon_0 \blacktriangleright F$. Hence, by definition, $\tau[t_\rho] \Vdash \Upsilon_0, F$.

• **Case** $\dfrac{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \qquad \Gamma \vdash \tau' : (\Upsilon_0; \Upsilon) \triangleright_\tau \Upsilon'}{\Gamma \vdash \tau\tau' : \Upsilon_0 \triangleright_\tau \Upsilon; \Upsilon'}\,(\tau;\tau')$. Let $\tau_0 \Vdash \Upsilon_0$. By induction hypothesis, we have that $\tau_0; \tau_\rho \Vdash \Upsilon_0; \Upsilon$ and thus $(\tau_0; \tau_\rho); \tau'_\rho \Vdash \Upsilon_0; \Upsilon; \Upsilon'$. Since τ_ρ is closed, we have that $\tau_0; (\tau_\rho; \tau'_\rho) \to_\tau (\tau_0; \tau_\rho); \tau'_\rho$ (by easy induction the structure of τ_ρ), hence $\tau_\rho; \tau'_\rho \Vdash \Upsilon_0 \triangleright_\tau \Upsilon; \Upsilon'$

• **Case** $(<:)$ rules. By definition, in all cases we have that σ_ρ is closed, hence $\vdash \sigma_\rho : \Upsilon'_\rho\,\mathtt{subst}\,\Upsilon\,\rho$ is well-typed, thus $\sigma_\rho \Vdash \Upsilon'_\rho\,\mathtt{subst}\,\Upsilon\,\rho$.

• **Case** $\dfrac{(\delta : \Upsilon \triangleright_\tau \Upsilon') \in \Gamma}{\Gamma \vdash \delta : \Upsilon \triangleright_\tau \Upsilon'}(\tau_{\mathsf{ax}})$. By definition, since $\rho \vdash \Gamma$ and $(\delta : \Upsilon \triangleright_\tau \Upsilon') \in \Gamma$, we have that $\rho(\delta) \Vdash \delta : \Upsilon_\rho \triangleright_\tau \Upsilon'_\rho$.

• **Case** $\dfrac{\Gamma, \delta : \Upsilon_0 \triangleright_\tau \Upsilon \vdash t : B}{\Gamma \vdash \lambda\delta.t : \Upsilon_0 \triangleright_\tau \Upsilon \to B}(\tau_I)$. Same proof as for the (λ) case.

• **Case** $\dfrac{\Gamma \vdash t : \Upsilon_0 \triangleright_\tau \Upsilon \to B \quad \Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon}{\Gamma \vdash t\,\tau : B}(\tau_E)$. Same proof as for the (@) case.

• **Case** $\dfrac{\Gamma, <: \Upsilon \vdash t : A \quad Y \notin FV(\Gamma)}{\Gamma \vdash \lambda Y.t : \forall Y.A}(\forall_I)$. Same proof as for the (λ) case.

• **Case** $\dfrac{\Gamma \vdash t : \forall Y.A}{\Gamma \vdash t\,\Upsilon : A\{Y := \Upsilon\}}(\forall_E)$. Same proof as for the (@) case.

• **Case** $\dfrac{\Gamma, s : Y <: \Upsilon \vdash t : A}{\Gamma \vdash \lambda s.t : Y <: \Upsilon \to A}(\forall_I)$. Same proof as for the (λ) case.

• **Case** $\dfrac{\Gamma \vdash t : \Upsilon' <: \Upsilon \to A \quad \Gamma \vdash \sigma : \Upsilon' <: \Upsilon}{\Gamma \vdash t\,\sigma : A}(\forall_E)$. Same proof as for the (@) case.

• **Case** $\dfrac{\begin{array}{c}\Gamma \vdash \tau : \Upsilon' \qquad \Gamma \vdash \sigma : \Upsilon' <: \Upsilon \qquad \Upsilon = [\![\Gamma_0]\!], F, [\![\Gamma_1]\!] \qquad |\Gamma_0| = n \\ \Gamma, s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \blacktriangleright F, s_1 : (Y_0, F; Y_1 <: \Upsilon), \delta_1 : (Y_0, F) \triangleright_\tau Y_1 \vdash t : A\end{array}}{\Gamma \vdash \mathtt{split}\ \tau\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon' <: \Upsilon\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ t : A}(\mathtt{split})$. By induction hypothesis, we get that:

1. $\tau_\rho \Vdash \Upsilon'_\rho$
2. $\sigma_\rho \Vdash \Upsilon'_\rho <: \Upsilon_\rho$
3. for any $\sigma_0 \Vdash Y_0 <: [\![\Gamma_0]\!]$, $\tau_0 \Vdash Y_0$, $u \Vdash Y_0 \blacktriangleright F$, $\sigma_1 \Vdash (Y_0, F; Y_1 <: \Upsilon)$, $\tau_1 \Vdash (Y_0, F) \triangleright_\tau Y_1$, if we define $\rho' = \rho, s_0 := \sigma_0, \delta_0 := \tau_0, x := u, s_1 := \sigma_1, \delta_1 := \tau_1$ then we have $t_{\rho'} \Vdash A$

In particular, by definition, we have that $\vdash \sigma_\rho : \Upsilon'_\rho <: \Upsilon_\rho$, hence Lemma 3.2 applies and we get that: $\sigma_\rho = \sigma_0^+; \sigma_1$ and $\Upsilon'_\rho = \Upsilon_0, F; \Upsilon_1$ with $\vdash \sigma_0 : \Upsilon_0 <: [\![\Gamma_0]\!]$ and $\Gamma \vdash \sigma : \Upsilon_1 <: [\![\Gamma_1]\!]$. Besides, since $\tau_\rho \Vdash \Upsilon_0, F; \Upsilon_1$, necessarily we have that $\tau_\rho = \tau_0[u]; \tau_1$ with $|\tau_0| = |\Upsilon_0|$, $\tau_0 \Vdash \Upsilon_0$, $\tau_1 \Vdash \Upsilon_1$ and $u \Vdash \Upsilon_0 \blacktriangleright F$ (this is a simple induction on the structure of τ_ρ). Therefore, if $\pi \in \|A\|$, we have:

$$\begin{array}{c}\mathtt{split}\ \tau_0[u]\tau_1\ \mathtt{at}\ n\ \mathtt{along}\ \sigma : \Upsilon_0, A, \Upsilon_1 <: [\![\Gamma_0]\!], A, [\![\Gamma_1]\!]\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ t \star \pi \\ \succ \\ t_\rho[\Upsilon_0/Y_0][\sigma_0/s_0][\tau_0/\delta_0][u/x][\Upsilon_1/Y_1][\tau_1/\delta_1][\sigma_1/s_1] \star \pi\end{array}$$

According to the third item above, this process belongs to the pole, and we can thus conclude by anti-reduction.

□

Proposition D.16. *The set $\perp\!\!\!\perp_\Downarrow = \{p : p \text{ normalizes}\}$ is a valid pole. Besides, the typing rule (c) is adequate for this pole*

Proof. Easy verification: if $p \succ p'$ and p' normalizes, then p normalizes too. Besides, for any $(\boldsymbol{k} : A) \in \mathcal{S}$, if $\pi \in \|A\|$ then $\boldsymbol{k} \star \pi$ is blocked hence belongs to $\perp\!\!\!\perp_\Downarrow$.

□

E Shallow embedding in Coq

We give here an overview of our take to define a shallow embedding of F_Υ in Coq. We define store types as list of types of the source calculus (Ty), stores through heterogeneous lists of terms, while the corresponding typing relation is defined as an inductive relation. Observe that we do treat the translation ▶ of types in the store as a parameter, emphasizing that the structure of F_Υ is actually orthogonal to the choice of this translation.

```
Parameter Ty:Type.

Inductive hlist:=
 |hnil : hlist
 |hcons : ∀ A, A → hlist → hlist.

Definition storeT := list Ty.

Inductive <: : storeT → storeT → Type:=
 | empty : nil <: nil
 | lift : ∀ Υ' Υ A, Υ'<: Υ → [Υ',A] <: Υ
 | plus : ∀ Υ' Υ A, Υ'<: Υ → [Υ',A] <: [Υ,A].
```

Coercions

```
Parameter ▶ : storeT → Ty → Type.

Context (F: storeT → Ty → Type)
        ( tr_sound:∀ Υ A, (Υ ▶ A) = ∀ Z, Z<:Υ → F Z A).

Definition store := hlist.

Inductive typedStore:store →storeT → storeT → Type :=
 | nilT : ∀ Y, typedStore hnil Y nil
 | consT : ∀ (A:Ty) (τ:store) (Υ0, Υ1:storeT) (t:Υ0; Υ1 ▶ A),
    typedStore τ Υ0 Υ1→ typedStore τ[t] Υ0 Υ1,A.
```

Stores and store types

Regarding coercions, we define them as the constructors of the inductive type defining the relation $<:$. In other words, we only define the coercions in normal form, since through the shallow embedding variables will be Coq variables, while $\sigma \circ \sigma'$, $\Uparrow_\Upsilon \sigma$ and $\sigma^{+\Upsilon}$ are easily definable as Coq functions. Similarly, $\uparrow^\sigma_\Upsilon \tau$ is also defined through its reduction rules, that is by induction Υ. That way, the reductions rules easily matches Coq reductions: for instance, observe that $\Uparrow_\Upsilon \sigma$ is defined by its reduction rules. Similarly, the only subtle reduction we have, to reduce `split` terms, will be precisely be the very definition of its embedding in Coq.

```
Fixpoint ⇑ Υ Υ0 Υ1 (σ:Υ1 <: Υ0):[Υ1;Υ]<:Υ0 := (* ⇑Υ σ *)
 match Υ with
  | nil ⇒ σ
  | [Υ',A] ⇒ lift A (⇑ Υ' σ)
  end.

Lemma liftN_lift {Υ1 Υ0 Υ} A (σ:Υ1 <: Υ0):
      (⇑Υ,A σ) = ⇑ (⇑Υ σ).
Proof.
  reflexivity.
Qed.

Definition ∘ : ∀ Υ2 Υ1 Υ0 (σ2:Υ2 <: Υ1) (σ1:Υ1 <: Υ0), Υ2 <: Υ0.
Proof
 (* Definition by induction on Υ2 *)
Defined


Lemma plus_lift {Υ2 Υ1 Υ0} A (σ1:Υ2<:Υ1) (σ0:Υ0<:Υ0):
      (σ1+) ∘ (⇑σ0) = ⇑( σ1 ∘ σ0 ).
Proof.
  reflexivity.
Qed.
```

```
Definition precomp_t (F:storeT → Type) Υ0 Υ1 (σ:Υ1<:Υ0)
  ( t:∀ Υ2, Υ2<:Υ0 → F Υ2) : (∀ Υ2, Υ2<:Υ1 → F Υ2)
 := ( fun Z (σ':Υ2<:Υ1) ⇒ t Z (σ' ∘ σ)).          (* ↑σ t *)

Definition presig_t Υ Υ' T (σ:Υ'<:Υ) (t:Υ ▶ T) : Υ' ▶ T.
rewrite tr_sound in *. apply (precomp_t _ σ). Defined.

Definition presig_τ Z :                          (* ↑σZ τ *)
    ∀ τ Υ0 Υ1 (σ:Υ1 <:Υ0) (H:typedStore τ Υ0 Z), store.
Proof.
 induction Z;intros;inversion H;subst.
 – exact hnil.
 – exact ([ IHZ τ0 Υ0 Υ1 σ X,presig_t (plusN Z σ) t]1).
Defined.

Lemma store_presig {A:Ty} ( Υ Υ0 Υ1: storeT) (τ : store)
( t :[ Υ0;Υ] ▶ A) (σ:Υ1 <: Υ0) (H:typedStore τ Υ0 Υ) :
  ↑σΥ,A τ[t] = (↑σΥ τ)[↑σ+Υ t].
Proof.
 intros. reflexivity.
Qed.
```

The full Coq development illustrating these ideas (and proving most of the properties we stated on F_Υ) is available as supplementary material.

F 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 F.1 (Store formation)**.** *The following rules are admissible:*

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

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

Proof. Straightforward typing derivations. For the left-hand side we have:

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

□

We then show that the construction lifting values to the level of terms is safe with respect to typing:

Lemma F.2 (Lifting values)**.** *The following is derivable:*

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

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

$$\dfrac{\dfrac{\Pi_E \qquad \dfrac{\Gamma \vdash V : \Upsilon \rhd_{\mathsf{V}} T \quad \overline{s : Y <: \Upsilon \vdash s : Y <: \Upsilon}\ (<:_{\mathsf{ax}})}{\Gamma, s : Y <: \Upsilon \vdash \uparrow^s V : Y \rhd_{\mathsf{V}} T}\ (\uparrow^\sigma)}{\Gamma, s : Y <: \Upsilon, \delta : Y, E : \Upsilon \rhd_{\mathsf{E}} T \vdash E\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s V) : \perp}\ (@)}{\Gamma \vdash \lambda Y s \delta E.E\ v\ \mathsf{id}_Y\ \delta\ (\uparrow^s V) : \Upsilon \rhd_{\mathsf{t}} T}\ (\lambda)$$

where we used Lemma D.9 and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{\overline{E : \Upsilon \rhd_{\mathsf{E}} T \vdash E : Y \rhd_{\mathsf{E}} T}\ (\mathrm{Ax})}{E : \Upsilon \rhd_{\mathsf{E}} T \vdash E\ Y : Y <: Y \to Y \to Y \rhd_{\mathsf{V}} T \to \perp}\ (\forall_E) \quad \dfrac{}{\vdash \mathsf{id}_Y : Y <: Y}\ (<:_{\mathsf{id}})}{\delta : Y, E : \Upsilon \rhd_{\mathsf{E}} T \vdash E\ Y\ \mathsf{id}_Y : Y \to Y \rhd_{\mathsf{V}} T \to \perp}\ (\forall_E) \quad \dfrac{}{\delta : Y \vdash \delta : Y}\ (\mathrm{Ax})}{\delta : Y, E : \Upsilon \rhd_{\mathsf{E}} T \vdash E\ Y\ \mathsf{id}_Y\ \delta : Y \rhd_{\mathsf{V}} T \to \perp}\ (@)$$

□

We are finally equipped to prove the main theorem:

Theorem 4.1. *The translation is well-typed, i.e.:*

1. *If* $\Gamma \vdash_v v : T$ *then* $[\![\Gamma \vdash_v v : T]\!]$
2. *If* $\Gamma \vdash_F F : T^\perp$ *then* $[\![\Gamma \vdash_F F : T^\perp]\!]$
3. *If* $\Gamma \vdash_V V : T$ *then* $[\![\Gamma \vdash_V V : T]\!]$
4. *If* $\Gamma \vdash_E E : T^\perp$ *then* $[\![\Gamma \vdash_E E : T^\perp]\!]$
5. *If* $\Gamma \vdash_t t : T$ *then* $[\![\Gamma \vdash_t t : T]\!]$
6. *If* $\Gamma \vdash_e e : T^\perp$ *then* $[\![\Gamma \vdash_e e : T^\perp]\!]$
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 D.8 to make the derivations more compact by systematically weakening contexts as soon as possible, and compact the first $(\forall_I)$ and (λ) rules in one rule.

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

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

$$\frac{\Gamma, x_i : T \vdash_t t : U \quad |\Gamma| = i}{\Gamma, \Gamma' \vdash_v \lambda x_i.t : T \to U}$$

Hence, we get by induction a proof Π_t of $[\![t]\!]_{\mathsf{t}} : [\![\Gamma, x_i : T]\!] \triangleright_{\mathsf{t}} U$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\vdash [\![t]\!]_t : \forall Y' <: [\![\Gamma, x_i : T]\!].Y' \to Y' \triangleright_{\mathsf{E}} U \to \bot} \quad \Pi_s}{s : Y <: [\![\Gamma, \Gamma']\!] \vdash [\![t]\!]_t\ (Y,T)\ \sigma_{\Gamma,\Gamma'} : (Y,T) \to (Y,T) \triangleright_{\mathsf{E}} U \to \bot}\ (\forall_E) \quad \Pi_\delta}{s : Y <: [\![\Gamma, \Gamma']\!], \delta : Y, u : Y \triangleright_{\mathsf{t}} T \vdash [\![t]\!]_t\ (Y,T)\ \sigma_{\Gamma,\Gamma'}\ \delta[u] : (Y,T) \triangleright_{\mathsf{E}} U \to \bot}\ (@) \quad \Pi_E}{s : Y <: [\![\Gamma, \Gamma']\!], \delta : Y, u : Y \triangleright_{\mathsf{t}} T, E : Y \triangleright_{\mathsf{E}} U \vdash [\![t]\!]_t\ (Y,T)\ \sigma_{\Gamma,\Gamma'}\ \delta[u]\ \uparrow^{\Uparrow \mathsf{id}_\Upsilon} E : \bot}\ (@)}{\vdash \lambda Y s \delta u E.[\![t]\!]_t\ (Y,T)\ \sigma_{\Gamma,\Gamma'}\ \delta[u]\ \uparrow^{\Uparrow \mathsf{id}_\Upsilon} E : \forall Y <: [\![\Gamma, \Gamma']\!].Y \to Y \triangleright_{\mathsf{t}} T \to Y \triangleright_{\mathsf{E}} U \to \bot}\ (\lambda)$$

where:

- Π_E is a proof of $E : Y \triangleright_{\mathsf{E}} U \vdash \uparrow^{\Uparrow \mathsf{id}_\Upsilon} E : (Y,T) \triangleright_{\mathsf{E}} U$ (derivable according to Lemma D.9);
- Π_δ is a proof of $\delta : Y, u : Y \triangleright_{\mathsf{t}} T; \vdash \delta[u] : Y, T$ (derivable according to Lemma F.1);
- $\sigma_{\Gamma,\Gamma'} \triangleq (s \circ \Uparrow_{[\![\Gamma']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+$ and Π_s is:

$$\dfrac{\dfrac{\dfrac{}{s : Y <: [\![\Gamma, \Gamma']\!] \vdash s : Y <: [\![\Gamma, \Gamma']\!]}\ (<:_{\mathsf{ax}}) \quad \dfrac{}{\vdash \mathsf{id}_{[\![\Gamma]\!]} : [\![\Gamma]\!] <: [\![\Gamma]\!]}}{\vdash \Uparrow_{[\![\Gamma']\!]} \mathsf{id}_{[\![\Gamma]\!]} : [\![\Gamma]\!], [\![\Gamma']\!] <: [\![\Gamma]\!]}\ (<:^{\Upsilon}_{\Uparrow})}{s : Y <: [\![\Gamma, \Gamma']\!] \vdash s \circ \Uparrow_{[\![\Gamma']\!]} \mathsf{id}_{[\![\Gamma]\!]} : Y <: [\![\Gamma]\!]}\ (<:_\circ)}{s : Y <: [\![\Gamma, \Gamma']\!] \vdash (s \circ \Uparrow_{[\![\Gamma']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+ : Y, T <: [\![\Gamma]\!], T}\ (<:_+)$$

2. Forcing contexts

• **Case** $[\![\kappa]\!]_{\mathsf{F}}$. $[\![\kappa]\!]_{\mathsf{F}} = \kappa$, which has the desired type by hypothesis.

• **Case** $[\![t.E]\!]_{\mathsf{F}}$. In the source language, we have:

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

Therefore, we have by induction hypothesis that $\vdash [\![t]\!]_{\mathsf{t}} : [\![\Gamma]\!] \triangleright_{\mathsf{t}} T$ and $\vdash [\![E]\!]_{\mathsf{t}} : [\![\Gamma]\!] \triangleright_{\mathsf{E}} U$, so that we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : Y \triangleright_{\mathsf{v}} T \to U \vdash v : \forall Y' <: Y : Y' \to Y' \triangleright_{\mathsf{t}} T \to Y' \triangleright_{\mathsf{E}} U \to \bot}\ (\mathrm{Ax}) \quad \Pi_s}{\delta : Y, v : Y \triangleright_{\mathsf{v}} T \to U \vdash v\ Y\ \mathsf{id}_Y : Y \to Y \triangleright_{\mathsf{t}} T \to Y \triangleright_{\mathsf{E}} U \to \bot}\ (\forall_I) \quad \Pi_\delta}{\delta : Y, v : Y \triangleright_{\mathsf{v}} T \to U \vdash v\ Y\ \mathsf{id}_Y\ \delta\ : Y \triangleright_{\mathsf{t}} T \to Y \triangleright_{\mathsf{E}} U \to \bot}\ (@) \quad \Pi_t}{s : Y <: [\![\Gamma]\!], \delta : Y, v : Y \triangleright_{\mathsf{v}} T \to U \vdash v\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s [\![t]\!]_{\mathsf{t}}) : Y \triangleright_{\mathsf{E}} U \to \bot}\ (@) \quad \Pi_E}{s : Y <: [\![\Gamma]\!], \delta : Y, v : Y \triangleright_{\mathsf{v}} T \to U \vdash v\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s [\![t]\!]_{\mathsf{t}})\ (\uparrow^s [\![E]\!]_{\mathsf{E}}) : \bot}\ (@)}{\vdash \lambda Y s \delta v.v\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s [\![t]\!]_{\mathsf{t}})\ (\uparrow^s [\![E]\!]_{\mathsf{E}}) : \forall Y <: [\![\Gamma]\!].Y \to Y \triangleright_{\mathsf{v}} T \to U \to \bot}\ (\lambda)$$

where:

- Π_E is a proof of $s : Y <: [\![\Gamma]\!] \vdash (\uparrow^s [\![E]\!]_{\mathsf{E}}) : Y \triangleright_{\mathsf{E}} U$, derived from the induction hypothesis for t and Lemma D.9;
- Π_t is a proof of $s : Y <: [\![\Gamma]\!] \vdash (\uparrow^s [\![t]\!]_{\mathsf{t}}) : Y \triangleright_{\mathsf{t}} T$, derived from the induction hypothesis for E and Lemma D.9;
- Π_δ is the axiom rule $\delta : Y \vdash \delta : Y$;
- Π_s is the proof of $\vdash \mathsf{id}_Y : Y <: Y$.

3. Weak values

• **Case** $[\![v]\!]_{\mathsf{V}}$. In the source language, we have:

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

Hence, we have by induction hypothesis that $\vdash [\![v]\!]_{\mathsf{v}} : [\![\Gamma]\!] \triangleright_{\mathsf{v}} T$ and we can derive:

$$\dfrac{\dfrac{\dfrac{F : Y \triangleright_{\mathsf{F}} T \vdash F : \forall Y' <: Y. Y' \to Y' \triangleright_{\mathsf{v}} T \to \bot \quad \Pi_Y}{s : Y <: [\![\Gamma]\!], F : Y \triangleright_{\mathsf{F}} T \vdash F\ Y\ \mathsf{id}_Y : Y \to Y \triangleright_{\mathsf{v}} T \to \bot}\,(@) \quad \dfrac{}{\delta : Y; \vdash \delta : Y}}{s : Y <: [\![\Gamma]\!], \delta : Y, F : Y \triangleright_{\mathsf{F}} T \vdash F\ Y\ \mathsf{id}_Y\ \delta : Y \triangleright_{\mathsf{v}} T \to \bot}\,(@) \quad \Pi_v}{s : Y <: [\![\Gamma]\!], \delta : Y, F : Y \triangleright_{\mathsf{F}} T \vdash F\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s [\![v]\!]_{\mathsf{v}}) : \bot}\,(@)}{\vdash \lambda Y s \delta F. F\ Y\ \mathsf{id}_Y\ \delta\ (\uparrow^s [\![v]\!]_{\mathsf{v}}) : \forall Y <: [\![\Gamma]\!]. Y \to Y \triangleright_{\mathsf{F}} T \to \bot}\,(\lambda)$$

where:

- Π_v is a proof of $s : Y <: [\![\Gamma]\!] \vdash \uparrow^s [\![v]\!]_{\mathsf{v}} : Y \triangleright_{\mathsf{v}} T$, derivable from the induction hypothesis and Lemma D.9.
- Π_δ is the axiom rule $\delta : Y \vdash \delta : Y$
- Π_Y is a proof of $\vdash \mathsf{id}_Y : Y <: Y$ (Proposition D.7)

• **Case** $[\![x_i]\!]_{\mathsf{V}}$. In the source language, we have:

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

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

$$\begin{aligned}[\![\Gamma_0, x_i : T, \Gamma_1 \vdash x_i : T]\!]_{\mathsf{V}} = &\lambda Y s \delta F.\ \mathtt{split}\ \delta\ \mathtt{at}\ i\ \mathtt{along}\,(s : Y <: [\![\Gamma_0]\!], T, [\![\Gamma_1]\!])\ \mathtt{as}\,(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\\ &\mathtt{in}\ x\ Y_0\ \mathsf{id}_{Y_0}\ \delta_0\ (\lambda Y_0' s_0' \delta_0' V. V\ (Y_0', T; Y_1)\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0'})\ (\delta_0'[\uparrow^{\mathsf{t}} V]; \uparrow^{s_0'^+} \delta_1)\ (\uparrow^{\sigma'} F))\end{aligned}$$

where $\uparrow^{\mathsf{t}} V = \lambda Z s \delta E. E\ Z\ \mathsf{id}_Z\ \delta\ (\uparrow^s V)$ and $\sigma' = (s'^+)^{+_{Y_1}}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : Y_0 \triangleright_{\mathsf{t}} T \vdash x : Y_0 \triangleright_{\mathsf{t}} T} \quad \dfrac{}{\vdash \mathsf{id}_{Y_0} : Y_0 <: Y_0}\,(<:_{\mathsf{ax}})}{x : Y_0 \triangleright_{\mathsf{t}} T \vdash x\ Y_0\ \mathsf{id}_{Y_0} : Y_0 \to Y_0 \triangleright_{\mathsf{E}} T \to \bot}\,(\forall_E) \quad \dfrac{}{\delta_0 : Y_0 \vdash \delta_0 : Y_0}\,(\mathrm{Ax})}{s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \triangleright_{\mathsf{t}} T \vdash x\ Y_0\ \mathsf{id}_{Y_0}\ \delta_0 : Y_0 \triangleright_{\mathsf{E}} T \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{F : Y_0, T; Y_1 \triangleright_{\mathsf{F}} T, s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \triangleright_{\mathsf{t}} T, s_1 : (Y_0, T; Y_1 <: \Upsilon), \delta_1 : (Y_0, T) \triangleright_\tau Y_1 \vdash x\ Y_0\ \mathsf{id}_{Y_0}\ \delta_0\ E : \bot}\,(@) \quad \boldsymbol{\Pi_s} \quad \boldsymbol{\Pi_\delta}}{(s : Y <: [\![\Gamma_0]\!], T, [\![\Gamma_1]\!]), \delta : Y, F : Y \triangleright_{\mathsf{F}} T \vdash \mathtt{split}\ \delta\ \mathtt{at}\ i\ \mathtt{along}\,(s : Y <: [\![\Gamma_0]\!], T, [\![\Gamma_1]\!])\ \mathtt{as}\,(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ x\ Y_0\ \mathsf{id}_{Y_0}\ \delta_0\ E : \bot}\,(\mathtt{split})}{\vdash \lambda Y s \delta F.\ \mathtt{split}\ \delta\ \mathtt{at}\ i\ \mathtt{along}\ s : Y <: [\![\Gamma_0]\!], T, [\![\Gamma_1]\!]\ \mathtt{as}\,(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ x\ Y_0\ \mathsf{id}_{Y_0}\ \delta_0\ E : \forall Y <: [\![\Gamma]\!]. Y \to Y \triangleright_{\mathsf{F}} T \to \bot}\,(\lambda)$$

where:

- Π_s is the axiom rule:

$$\dfrac{}{s : Y <: [\![\Gamma_0]\!], T, [\![\Gamma_1]\!] \vdash s : Y <: [\![\Gamma_0]\!], T; [\![\Gamma_1]\!]}\,(<:_{\mathsf{ax}})$$

- Π_δ is the axiom rule:

$$\dfrac{}{\delta : Y \vdash \delta : Y}\,(\tau_{\mathsf{ax}})$$

- $E = \lambda Y_0' s_0' \delta_0' V. V\ (Y_0', T; Y_1)\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0'})\ (\delta_0'[\uparrow^{\mathsf{t}} V]; \uparrow^{s_0'^+} \delta_1)\ (\uparrow^{\sigma'} F)$ and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_{\mathsf{V}} T; \vdash V : Y_0' \triangleright_{\mathsf{t}} T}\,(\mathrm{Ax}) \quad \Gamma \vdash \Uparrow_{Y_1} \mathsf{id}_{Y_0'} : Y_0', T, Y_1 <: Y_0'}{\Gamma, V : Y_0' \triangleright_{\mathsf{V}} T \vdash V\ (Y_0', T; Y_1)\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0'}) : (Y_0', T, Y_1) \to (Y_0', T, Y_1) \triangleright_{\mathsf{F}} T \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi'_\tau}}{\Gamma, s_0' : Y_0' <: Y_0, V : Y_0' \triangleright_{\mathsf{V}} T \vdash V\ (Y_0', T; Y_1)\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0'})\ (\delta_0'[\uparrow^{\mathsf{t}} V]; \uparrow^{s_0'^+} \delta_1) : (Y_0', T, Y_1) \triangleright_{\mathsf{F}} T \to \bot}\,(@) \quad \boldsymbol{\Pi_f}}{\Gamma, s_0' : Y_0' <: Y_0, \delta_0' : Y_0', V : Y_0' \triangleright_{\mathsf{V}} T \vdash V\ (Y_0', T; Y_1)\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0'})\ (\delta_0'[\uparrow^{\mathsf{t}} V]; \uparrow^{s_0'^+} \delta_1)\ (\uparrow^{\sigma'} F) : \bot}\,(@)}{\Gamma \vdash \lambda Y_0' s_0' \delta_0' V. V\ (Y_0', T; Y_1)\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0'})\ (\delta_0'[\uparrow^{\mathsf{t}} V]; \uparrow^{s_0'^+} \delta_1)\ (\uparrow^{\sigma'} F) : Y_0 \triangleright_{\mathsf{E}} T}\,(\lambda)$$

  where $\Gamma = F : Y_0, T; Y_1 \triangleright_{\mathsf{F}} T, s_0 : Y_0 <: [\![\Gamma_0]\!], \delta_0 : Y_0, x : Y_0 \triangleright_{\mathsf{t}} T, s_1 : (Y_0, T; Y_1 <: \Upsilon), \delta_1 : (Y_0, T) \triangleright_\tau Y_1$.

- Π_F is the following proof, obtained by Lemma D.9:

$$\dfrac{\dfrac{}{F : (Y_0, T, Y_1) \triangleright_{\mathsf{F}} T; \vdash F : (Y_0, T, Y_1) \triangleright_{\mathsf{F}} T}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{s_0' : Y_0' <: Y_0 \vdash s_0' : Y_0' <: Y_0}\,(<:_{\mathsf{ax}})}{s_0' : Y_0' <: Y_0 \vdash (s_0'^+)^{+_{Y_1}} : Y_0', T, Y_1 <: Y_0, T, Y_1}\,(<:_+^{\Upsilon})}{F : (Y_0, T, Y_1) \triangleright_{\mathsf{F}} T; s_0' : Y_0' <: Y_0^n \vdash (\uparrow^{\sigma'} F) : (Y_0', T, Y_1) \triangleright_{\mathsf{F}} T}\,(\uparrow^\sigma)$$

- Π'_τ is the following derivation, where we use Lemmas F.1 and F.2:

$$\dfrac{\dfrac{\dfrac{}{\delta'_0 : Y'_0 \vdash \delta'_0 : Y'_0}\,(\tau_{\mathsf{ax}}) \quad \dfrac{}{V : Y'_0 \triangleright_{\mathsf{V}} T \vdash \uparrow^{\mathsf{t}} V : Y'_0 \triangleright_{\mathsf{t}} T}\,(\uparrow^{\mathsf{t}})}{Y'_0 <: Y_0, \delta'_0 : Y'_0, V : Y'_0 \triangleright_{\mathsf{V}} T \vdash \delta'_0[\uparrow^{\mathsf{t}} V] : Y'_0, T}\,(\tau[t]) \quad \dfrac{\dfrac{}{\delta_1 : (Y_0, T) \triangleright_\tau Y_1 \vdash \delta_1 : (Y_0, T) \triangleright_\tau Y_1}\,(\tau_{\mathsf{ax}}) \quad \dfrac{\dfrac{}{s'_0 : Y'_0 <: Y_0 \vdash s : Y'_0 <: Y_0}\,(<:_{\mathsf{ax}})}{s'_0 : Y'_0 <: Y_0 \vdash s'^{+}_0 : Y'_0, T <: Y_0, T}\,(<:_+)}{\delta_1 : (Y_0, T) \triangleright_\tau Y_1; s'_0 : Y'_0 <: Y_0 \vdash \uparrow^{s'^{+}_0} \delta_1 : Y'_0, T \triangleright_\tau Y_1}\,(\tau_{<:})}{\delta_1 : (Y_0, T) \triangleright_\tau Y_1, s'_0 : Y'_0 <: Y_0, \delta'_0 : Y'_0, V : Y'_0 \triangleright_{\mathsf{V}} T \vdash \delta'_0[\uparrow^{\mathsf{t}} V]; \uparrow^{s'^{+}_0} \delta_1 : Y'_0, T, Y_1}$$

4. Catchable contexts

• **Case** $[\![F]\!]_{\mathsf{E}}$. This case is similar to the case $[\![v]\!]_{\mathsf{V}}$.

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

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

We have by induction hypothesis a proof of $\vdash [\![\tau']\!]_\tau : [\![\Gamma, x_i : T]\!] \triangleright_\tau [\![\Gamma']\!]$ and a proof Π_F of $\vdash [\![F]\!]_{\mathsf{F}} : [\![\Gamma, x_i : T, \Gamma']\!] \triangleright_{\mathsf{F}} T$.
We can thus derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y \triangleright_{\mathsf{V}} T \vdash V : Y \triangleright_{\mathsf{V}} T}\,(\mathrm{Ax}) \quad \dfrac{}{\vdash \Uparrow_{T;[\![\Gamma']\!]} \mathsf{id}_\Upsilon : Y, T; [\![\Gamma']\!] <: Y}}{V : Y \triangleright_{\mathsf{V}} T \vdash V\ (\Upsilon, T; [\![\Gamma']\!])\ \sigma_V : (Y, T, [\![\Gamma']\!]) \to (Y, T, [\![\Gamma']\!]) \triangleright_{\mathsf{F}} T \to \perp}\,(\forall_E) \quad \boldsymbol{\Pi}_\tau}{s : Y <: [\![\Gamma, \Gamma'']\!], \delta : Y, V : Y \triangleright_{\mathsf{V}} T \vdash V\ (\Upsilon, T; [\![\Gamma']\!])\ \sigma_V\ (\delta[\uparrow^t V]_t; \uparrow^{\sigma_\tau} [\![\tau']\!]_\tau) : (Y, T, [\![\Gamma']\!]) \triangleright_{\mathsf{F}} T \to \perp}\,(@) \quad \boldsymbol{\Pi}_F}{s : Y <: [\![\Gamma, \Gamma'']\!], \delta : Y, V : Y \triangleright_{\mathsf{V}} T \vdash V\ (\Upsilon, T; [\![\Gamma']\!])\ \sigma_V\ (\delta[\uparrow^t V]_t; \uparrow^{\sigma_\tau} [\![\tau']\!]_\tau)\ (\uparrow^{\sigma_F} [\![F]\!]_{\mathsf{F}}) : \perp}\,(@)}{\vdash \lambda Y s \delta V.V\ (\Upsilon, T; [\![\Gamma']\!])\ \sigma_V\ (\delta[\uparrow^t V]_t; \uparrow^{\sigma_\tau} [\![\tau']\!]_\tau)\ (\uparrow^{\sigma_F} [\![F]\!]_{\mathsf{F}}) : [\![\Gamma, \Gamma'']\!] \triangleright_{\mathsf{F}} T}\,(\lambda)$$

where:

- $\sigma_\tau = (s \circ \Uparrow_{[\![\Gamma'']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+$, $\sigma_F = \sigma_\tau^{+_{[\![\Gamma']\!]}}$ and $\sigma_V = \Uparrow_{A;[\![\Gamma']\!]} \mathsf{id}_\Upsilon$
- Π_F is the following proof, obtained by Lemma D.9:

$$\dfrac{\vdash F : ([\![\Gamma]\!], T; [\![\Gamma']\!]) \triangleright_{\mathsf{F}} T \quad \dfrac{\dfrac{\dfrac{}{s : Y <: [\![\Gamma, \Gamma'']\!] \vdash s : Y <: [\![\Gamma, \Gamma'']\!]}\,(<:_{\mathsf{ax}}) \quad \dfrac{\vdash \mathsf{id}_{[\![\Gamma]\!]} : [\![\Gamma]\!] <: [\![\Gamma]\!]}{\vdash \Uparrow_{[\![\Gamma'']\!]} \mathsf{id}_{[\![\Gamma]\!]} : [\![\Gamma, \Gamma'']\!] <: [\![\Gamma]\!]}\,(<:^\Upsilon_{\Uparrow})}{s : Y <: [\![\Gamma, \Gamma'']\!] \vdash s \circ \Uparrow_{[\![\Gamma'']\!]} \mathsf{id}_{[\![\Gamma]\!]} : Y <: [\![\Gamma]\!]}\,(<:_\circ)}{s : Y <: [\![\Gamma, \Gamma'']\!] \vdash ((s \circ \Uparrow_{[\![\Gamma'']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+)^{+_{[\![\Gamma']\!]}} : Y, T; [\![\Gamma']\!] <: [\![\Gamma]\!], T, [\![\Gamma']\!]}\,(<:^\Upsilon_+)}{s : Y <: [\![\Gamma, \Gamma'']\!] \vdash (\uparrow^{\sigma_F} F) : (Y, T, [\![\Gamma']\!]) \triangleright_{\mathsf{F}} T}\,(\uparrow^\sigma)$$

- Π_τ is the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\delta : Y \vdash \delta : Y}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{V : Y \triangleright_{\mathsf{V}} T \vdash V : Y \triangleright_{\mathsf{V}} T}\,(\mathrm{Ax})}{V : Y \triangleright_{\mathsf{V}} T \vdash \uparrow^{\mathsf{t}} V : Y \triangleright_{\mathsf{t}} T}\,(\uparrow^{\mathsf{t}})}{\delta : Y, V : Y \triangleright_{\mathsf{V}} T \vdash \delta[\uparrow^{\mathsf{t}} V]_t : Y, T}\,(\tau[t]) \quad \dfrac{\vdash [\![\tau']\!]_\tau : [\![\Gamma]\!], T \triangleright_\tau [\![\Gamma']\!] \quad \dfrac{\text{see } \Pi_F}{s : Y <: [\![\Gamma]\!] \vdash ((s \circ \Uparrow_{[\![\Gamma'']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+) : Y, T <: [\![\Gamma]\!], T}\,(<:_+)}{s : Y <: [\![\Gamma, \Gamma'']\!] \vdash \uparrow^{s^+} [\![\tau']\!]_\tau : Y, T \triangleright_\tau [\![\Gamma']\!]}\,(\uparrow\tau)}{s : Y <: [\![\Gamma, \Gamma'']\!], \delta : Y, V : Y \triangleright_{\mathsf{V}} T \vdash (\delta[\uparrow^{\mathsf{t}} V]_t; \uparrow^{\sigma_\tau} [\![\tau']\!]_\tau)}\,(\tau;\tau')$$

5. Terms

• **Case** $[\![V]\!]_{\mathsf{t}}$. This case is similar to the case $[\![v]\!]_{\mathsf{V}}$.

• **Case** $[\![\mu\alpha_i.c]\!]_{\mathsf{t}}$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\dfrac{\Gamma, \alpha_i : T^\perp \vdash_c c \quad |\Gamma| = i}{\Gamma, \Gamma' \vdash_t \mu\alpha_i.c : T}$$

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

$$\dfrac{\vdash [\![c]\!]_{\mathrm{c}} : [\![\Gamma, x_i : T^{\perp}]\!] \triangleright_{\mathrm{c}} \perp \qquad \dfrac{\dfrac{\dfrac{\Pi_s}{s : Y <: [\![\Gamma, \Gamma']\!] \vdash (s \circ \Uparrow_{[\![\Gamma']\!]} \mathrm{id}_{[\![\Gamma]\!]})^{+} : Y, T <: [\![\Gamma]\!], T}\,(<:_{+})}{s : Y <: [\![\Gamma, \Gamma']\!], \delta : Y \vdash [\![c]\!]_{\mathrm{c}}\ (\Upsilon, A^{\perp})\ \sigma_{\Gamma,\Gamma'} : (Y, T^{\perp}) \to \perp}\,(\forall_E) \qquad \dfrac{\dfrac{}{\delta : Y \vdash \delta : \triangleright_{\tau} Y}\,(\mathrm{Ax}) \qquad \dfrac{}{E : Y \triangleright_{\mathrm{E}} T \vdash E : Y \triangleright_{\mathrm{E}} T}\,(\mathrm{Ax})}{\delta : Y, E : Y \triangleright_{\mathrm{E}} T \vdash \delta[E] : \triangleright_{\tau} Y, T^{\perp}}\,(\tau[E])}{s : Y <: [\![\Gamma, \Gamma']\!], \delta : Y, E : Y \triangleright_{\mathrm{E}} T \vdash [\![c]\!]_{\mathrm{c}}\ (\Upsilon, A^{\perp})\ \sigma_{\Gamma,\Gamma'}\ \delta[E]_E : \perp}\,(@)}{\vdash \lambda Y s \delta E.[\![c]\!]_{\mathrm{c}}\ (\Upsilon, A^{\perp})\ \sigma_{\Gamma,\Gamma'}\ \delta[E] : [\![\Gamma, \Gamma']\!] \triangleright_{\mathrm{t}} T}\,(\lambda)$$

where Π_s is as in the first case for $\lambda x_i.t$.

6. Contexts

• **Case** $[\![E]\!]_{\mathrm{e}}$. This case is similar to the case $[\![v]\!]_{\mathrm{V}}$.

• **Case** $[\![\tilde{\mu} x_i.c]\!]_{\mathrm{e}}$. This case is similar to the case $[\![\mu \alpha_i.c]\!]_{\mathrm{t}}$.

7. Commands

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

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

thus we get by induction two proofs of $;\vdash [\![t]\!]_{\mathrm{t}} : [\![\Gamma]\!] \triangleright_{\mathrm{t}} T$ and $;\vdash [\![e]\!]_{\mathrm{c}} : [\![\Gamma]\!] \triangleright_{\mathrm{e}} T$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\vdash [\![e]\!]_{\mathrm{e}} : [\![\Gamma]\!] \triangleright_{\mathrm{e}} T \quad \dfrac{}{s : Y <: [\![\Gamma]\!] \vdash s : Y <: [\![\Gamma]\!]}\,(<:_{\mathrm{ax}})}{s : Y <: [\![\Gamma]\!], \delta : Y \vdash [\![e]\!]_{\mathrm{e}}\ Y\ s : Y \to Y \triangleright_{\mathrm{t}} T \to \perp}\,(\forall_E) \qquad \dfrac{}{\delta : Y \vdash \delta : Y}\,(\mathrm{Ax})\,(\tau_E)}{s : Y <: [\![\Gamma]\!], \delta : Y \vdash [\![e]\!]_{\mathrm{e}}\ Y\ s\ \delta : Y \triangleright_{\mathrm{t}} T \to \perp} \qquad \dfrac{\vdash [\![t]\!]_{\mathrm{t}} : [\![\Gamma]\!] \triangleright_{\mathrm{t}} T \quad \dfrac{}{s : Y <: [\![\Gamma]\!] \vdash s : Y <: [\![\Gamma]\!]}\,(<:_{\mathrm{ax}})}{s : Y <: [\![\Gamma]\!] \vdash \uparrow^{s} [\![t]\!]_{\mathrm{t}} : Y \triangleright_{\mathrm{t}} T}\,(@)}{s : Y <: [\![\Gamma]\!], \delta : Y \vdash [\![e]\!]_{\mathrm{e}}\ Y\ s\ \delta\ (\uparrow^{s} [\![t]\!]_{\mathrm{t}}) : \perp}\,(\lambda)}{\vdash \lambda Y s \delta.[\![e]\!]_{\mathrm{e}}\ Y\ s\ \delta\ (\uparrow^{s} [\![t]\!]_{\mathrm{t}}) : [\![\Gamma]\!] \triangleright_{\mathrm{c}} \perp}$$

8. Closures

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

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

We thus get by induction two proofs $\vdash [\![\tau']\!]_{\tau} : [\![\Gamma]\!] \triangleright_{\tau} [\![\Gamma']\!]$ and $\vdash [\![c]\!]_{\mathrm{c}} : [\![\Gamma, \Gamma']\!] \triangleright_{\mathrm{c}} \perp$. We can derive:

$$\dfrac{\vdash [\![c]\!]_{\mathrm{c}} : [\![\Gamma, \Gamma']\!] \triangleright_{\mathrm{c}} \perp \quad \dfrac{\dfrac{\dfrac{}{s : Y <: [\![\Gamma]\!] \vdash s : Y <: [\![\Gamma]\!]}\,(<:_{\mathrm{ax}})}{s : Y <: [\![\Gamma]\!] \vdash (s^{+[\![\Gamma']\!]}) : (Y; [\![\Gamma']\!]) <: [\![\Gamma, \Gamma']\!]}\,(<:^{\Upsilon}_{+})}{s : Y <: [\![\Gamma]\!] \vdash [\![c]\!]_{\mathrm{c}}\ (Y; [\![\Gamma']\!])\ (s^{+[\![\Gamma']\!]}) : (Y, [\![\Gamma']\!]) \to \perp}\,(\forall_E) \qquad \dfrac{\dfrac{}{\delta : Y \vdash \delta : Y}\,(\mathrm{Ax}) \qquad \dfrac{\vdash [\![\tau']\!]_{\tau} : [\![\Gamma]\!] \triangleright_{\tau} [\![\Gamma']\!] \quad \dfrac{}{\vdash s : Y <: [\![\Gamma]\!]}\,(<:_{\mathrm{ax}})}{\vdash \uparrow^{s} [\![\tau']\!]_{\tau} : Y \triangleright_{\tau} [\![\Gamma']\!]}\,(\uparrow\tau)}{s : Y <: [\![\Gamma]\!] \vdash \delta; \uparrow^{s} [\![\tau']\!]_{\tau} : Y [\![\Gamma']\!]}\,(\tau;\tau')}{s : Y <: [\![\Gamma]\!], \delta : Y \vdash [\![c]\!]_{\mathrm{c}}\ (Y; [\![\Gamma']\!])\ (s^{+[\![\Gamma']\!]})\ (\delta; \uparrow^{s'} [\![\tau']\!]_{\tau}) : \perp}\,(\tau_E)}{\vdash \lambda Y s \delta.[\![c]\!]_{\mathrm{c}}\ (Y; [\![\Gamma']\!])\ (s^{+[\![\Gamma']\!]})\ (\delta; \uparrow^{s} [\![\tau']\!]_{\tau})}\,(\lambda)$$

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:

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

By induction, we obtain two proofs of $\vdash [\![\tau]\!]_{\tau} : [\![\Gamma]\!] \triangleright_{\tau} [\![\Gamma']\!]$ and $\vdash [\![t]\!]_{\mathrm{t}} : [\![\Gamma, \Gamma']\!] \triangleright_{\mathrm{t}} T$. We can thus derive by Lemma F.1:

$$\dfrac{\vdash [\![\tau]\!]_{\tau} : [\![\Gamma]\!] \triangleright_{\tau} [\![\Gamma']\!] \quad \vdash [\![t]\!]_{\mathrm{t}} : [\![\Gamma, \Gamma']\!] \triangleright_{\mathrm{t}} T}{\vdash [\![\tau]\!]_{\tau} [[\![t]\!]_{\mathrm{t}}] : [\![\Gamma]\!] \triangleright_{\tau} [\![\Gamma']\!], T}\,(\tau[t])$$

□

$$\llbracket \Gamma_i, \Gamma' \vdash \lambda x_i.t : A \to B \rrbracket_{\mathsf{V}}\ \Upsilon\ \sigma\ \tau\ u\ E \triangleq \llbracket t \rrbracket_t\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[u]\ \uparrow^{\Uparrow \mathrm{id}_\Upsilon} E \qquad \llbracket \boldsymbol{k} \rrbracket_{\mathsf{V}} \triangleq \boldsymbol{k}$$

$$\llbracket t \cdot E \rrbracket_{\mathsf{E}}\ \Upsilon\ \sigma\ \tau\ v \triangleq v\ \Upsilon\ \mathrm{id}_\Upsilon\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_{\mathsf{t}})\ (\uparrow^\sigma \llbracket E \rrbracket_{\mathsf{E}}) \qquad \llbracket \boldsymbol{\kappa} \rrbracket_{\mathsf{E}} \triangleq \boldsymbol{\kappa}$$

$$\llbracket \alpha_i \rrbracket_{\mathsf{E}}\ \Upsilon\ \sigma\ \tau\ V \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\,(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\,\mathtt{in}\,x\ \Upsilon\ (\Uparrow\!\!\Uparrow_{Y_1} \mathrm{id}_{Y_0})\ \tau\ V$$

$$\llbracket V \rrbracket_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ E \triangleq E\ \Upsilon\ \mathrm{id}_\Upsilon\ \tau\ (\uparrow^\sigma \llbracket V \rrbracket_{\mathsf{V}}) \qquad \llbracket \Gamma_i, \Gamma' \vdash \mu\alpha_i.c : A \rrbracket_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ E \triangleq \llbracket c \rrbracket_{\mathsf{c}}\ (\Upsilon, A^\perp)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[E]$$

$$\llbracket x_i \rrbracket_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ E \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\,(Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\,\mathtt{in}\,x\ \Upsilon\ (\Uparrow\!\!\Uparrow_{Y_1} \mathrm{id}_{Y_0})\ \tau\ E$$

$$\llbracket E \rrbracket_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ t \triangleq t\ \Upsilon\ \mathrm{id}_\Upsilon\ \tau\ (\uparrow^\sigma \llbracket E \rrbracket_{\mathsf{E}}) \qquad \llbracket \Gamma_i, \Gamma' \vdash \tilde{\mu}x_i.c : A^\perp \rrbracket_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ t \triangleq \llbracket c \rrbracket_{\mathsf{c}}\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[t]$$

$$\llbracket \langle t \parallel e \rangle \rrbracket_{\mathsf{c}}\ \Upsilon\ \sigma\ \tau \triangleq \llbracket e \rrbracket_{\mathsf{e}}\ \ \Upsilon\ \sigma\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_{\mathsf{t}}) \qquad \llbracket c\tau' \rrbracket_{\mathsf{l}}\ \Upsilon\ \sigma\ \tau \triangleq \llbracket c \rrbracket_{\mathsf{c}}\ (\Upsilon; \Gamma')\ (\sigma^{+\llbracket \Gamma' \rrbracket})\ (\tau; \uparrow^\sigma \llbracket \tau' \rrbracket_\tau) \quad \textit{where}^\dagger\ \tau' : \Gamma'$$

$$\llbracket \varepsilon \rrbracket_\tau \triangleq \varepsilon \qquad \llbracket \tau_0[x_i := t] \rrbracket_\tau \triangleq \llbracket \tau_0 \rrbracket_\tau[\llbracket t \rrbracket_{\mathsf{t}}] \qquad \llbracket \tau_0[\alpha_i := E] \rrbracket_\tau \triangleq \llbracket \tau_0 \rrbracket_\tau[\llbracket E \rrbracket_{\mathsf{E}}]$$

where $\mathrm{id}_\Upsilon = \varepsilon^{+\Upsilon}$, $\sigma_{\Upsilon,\Gamma'} \triangleq (\sigma \circ \Uparrow_{\llbracket \Gamma' \rrbracket} \mathrm{id}_{\llbracket \Gamma \rrbracket})^+$ and Γ_i indicates that $|\Gamma| = i$

$^\dagger\tau' : \Gamma'$ in the source of the translation

(a) Translation of terms

$$\begin{array}{lll}
\llbracket \Gamma \vdash_e e : T^\perp \rrbracket \triangleq \vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket \rhd_{\mathsf{e}} T & \llbracket \Gamma \vdash_c c \rrbracket \triangleq \vdash \llbracket c \rrbracket_c : \llbracket \Gamma \rrbracket \rhd_{\mathsf{c}} \perp & \llbracket \varepsilon \rrbracket \triangleq \varepsilon \\
\llbracket \Gamma \vdash_t t : T \rrbracket \triangleq \vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket \rhd_{\mathsf{t}} T & \llbracket \Gamma \vdash_l l \rrbracket \triangleq \vdash \llbracket l \rrbracket_l : \llbracket \Gamma \rrbracket \rhd_{\mathsf{c}} \perp & \llbracket \Gamma, x_i : T \rrbracket \triangleq \llbracket \Gamma \rrbracket, T \\
\llbracket \Gamma \vdash_E E : T^\perp \rrbracket \triangleq \vdash \llbracket E \rrbracket_E : \llbracket \Gamma \rrbracket \rhd_{\mathsf{E}} T & \llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket \triangleq \vdash \llbracket \tau \rrbracket_\tau : \llbracket \Gamma \rrbracket \rhd_\tau \llbracket \Gamma' \rrbracket & \llbracket \Gamma, \alpha_i : T^\perp \rrbracket \triangleq \llbracket \Gamma \rrbracket, T^\perp \\
\llbracket \Gamma \vdash_V V : T \rrbracket \triangleq \vdash \llbracket V \rrbracket_V : \llbracket \Gamma \rrbracket \rhd_{\mathsf{V}} T &&
\end{array}$$

$$\begin{array}{ll}
\Upsilon \rhd_{\mathsf{c}} T \triangleq \forall Y <: \Upsilon.\, Y \to \perp & \Upsilon \rhd_{\mathsf{V}} X \triangleq X \\
\Upsilon \rhd_{\mathsf{e}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_{\mathsf{t}} T) \to \perp & \Upsilon \rhd_{\mathsf{V}} T \to U \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_{\mathsf{t}} T) \to (Y \rhd_{\mathsf{E}} U) \to \perp \\
\Upsilon \rhd_{\mathsf{t}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_{\mathsf{E}} T) \to \perp & \Upsilon \blacktriangleright T \triangleq \Upsilon \rhd_{\mathsf{t}} T \\
\Upsilon \rhd_{\mathsf{E}} T \triangleq \forall Y <: \Upsilon.\, Y \to (Y \rhd_{\mathsf{V}} T) \to \perp & \Upsilon \blacktriangleright T^\perp \triangleq \Upsilon \rhd_{\mathsf{E}} T
\end{array}$$

(b) Translation of types and judgments

Figure 17. Call-by-name continuation-and-environment-passing style translation

G A typed call-by-name translation

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

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

We give in Figure 17 the full translation for the call-by-name $\bar{\lambda}\mu\tilde{\mu}$-calculus with global environment. Once again, we have:

Theorem 4.2. *The translation is well-typed,* i.e.*:*

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

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 Appendix F 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 $\llbracket t \rrbracket_{\mathsf{t}}$ 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 G.1. Consider a term $x : A, y : B \vdash_t u : C$, through the translation we will thus have $\vdash [\![u]\!]_{\mathrm{t}} : A, B \triangleright_{\mathrm{t}} C \to D$. 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 $\tau_i : \vec{U}_i$) 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 $\vec{U}_0, A, \vec{U}_1, B, \vec{U}_2$.

$$[\![\Gamma_i, \Gamma' \vdash \lambda x_i.t : A \to B]\!]_{\mathsf{V}}\ \Upsilon\ \sigma\ \tau\ u\ E \triangleq [\![t]\!]_t\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[u]\ \uparrow^{\Uparrow \mathsf{id}_\Upsilon} E \qquad [\![\boldsymbol{k}]\!]_{\mathsf{V}} \triangleq \boldsymbol{k}$$
$$[\![x_i]\!]_{\mathsf{V}}\ \Upsilon\ \sigma\ \tau \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ x\ \Upsilon\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0})\ \tau$$

$$[\![t \cdot E]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ V \triangleq V\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![t]\!]_{\mathsf{t}})\ (\uparrow^\sigma [\![E]\!]_{\mathsf{E}}) \qquad [\![\boldsymbol{\kappa}]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ V \triangleq V\ \Upsilon\ \mathsf{id}_\Upsilon\ \boldsymbol{\kappa}$$
$$[\![\alpha_i]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ V \triangleq \mathtt{split}\ \tau\ \mathtt{at}\ i\ \mathtt{along}\ \sigma\ \mathtt{as}\ (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1)\ \mathtt{in}\ x\ \Upsilon\ (\Uparrow_{Y_1} \mathsf{id}_{Y_0})\ \tau\ V$$
$$[\![\Gamma_i, \Gamma' \vdash \tilde{\mu} x_i.c : A^\perp]\!]_{\mathsf{e}}\ \Upsilon\ \sigma\ \tau\ V \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon, A)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[V]$$

$$[\![V]\!]_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ e \triangleq e\ \Upsilon\ \mathsf{id}_\Upsilon\ \tau\ (\uparrow^\sigma [\![V]\!]_{\mathsf{V}})$$
$$[\![\Gamma_i, \Gamma' \vdash \mu\alpha_i.c : A]\!]_{\mathsf{t}}\ \Upsilon\ \sigma\ \tau\ e \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon, A^\perp)\ \sigma_{\Gamma_i,\Gamma'}\ \tau[e]$$

$$[\![\langle t \parallel e \rangle]\!]_{\mathsf{c}}\ \Upsilon\ \sigma\ \tau \triangleq [\![t]\!]_{\mathsf{t}}\ \ \Upsilon\ \sigma\ \tau\ (\uparrow^\sigma [\![e]\!]_{\mathsf{e}}) \qquad [\![c\tau']\!]_{\mathsf{l}}\ \Upsilon\ \sigma\ \tau \triangleq [\![c]\!]_{\mathsf{c}}\ (\Upsilon; \Gamma')\ (\sigma^{+[\![\Gamma']\!]})\ (\tau; \uparrow^\sigma [\![\tau']\!]_\tau) \qquad \textit{where}^\dagger\ \tau' : \Gamma'$$

$$[\![\varepsilon]\!]_\tau \triangleq \varepsilon \qquad [\![\tau_0[x_i := V]]\!]_\tau \triangleq [\![\tau_0]\!]_\tau[[\![V]\!]_{\mathsf{V}}] \qquad [\![\tau_0[\alpha_i := e]]\!]_\tau \triangleq [\![\tau_0]\!]_\tau[[\![e]\!]_{\mathsf{e}}]$$

where $\mathsf{id}_\Upsilon = \varepsilon^{+\Upsilon}$, $\sigma_{\Gamma,\Gamma'} \triangleq (\sigma \circ \Uparrow_{[\![\Gamma']\!]} \mathsf{id}_{[\![\Gamma]\!]})^+$ and Γ_i indicates that $|\Gamma| = i$

$^\dagger\tau' : \Gamma'$ in the source of the translation

(a) Translation of terms

$$[\![\Gamma \vdash_t t : T]\!] \triangleq \vdash [\![t]\!]_t : [\![\Gamma]\!] \triangleright_{\mathsf{t}} T \qquad [\![\Gamma \vdash_c c]\!] \triangleq \vdash [\![c]\!]_c : [\![\Gamma]\!] \triangleright_{\mathsf{c}} \perp \qquad [\![\varepsilon]\!] \triangleq \varepsilon$$
$$[\![\Gamma \vdash_e e : T^\perp]\!] \triangleq \vdash [\![e]\!]_e : [\![\Gamma]\!] \triangleright_{\mathsf{e}} T \qquad [\![\Gamma \vdash_l l]\!] \triangleq \vdash [\![l]\!]_l : [\![\Gamma]\!] \triangleright_{\mathsf{c}} \perp \qquad [\![\Gamma, x_i : T]\!] \triangleq [\![\Gamma]\!], T$$
$$[\![\Gamma \vdash_V V : T]\!] \triangleq \vdash [\![V]\!]_V : [\![\Gamma]\!] \triangleright_{\mathsf{V}} T \qquad [\![\Gamma \vdash_\tau \tau : \Gamma']\!] \triangleq \vdash [\![\tau]\!]_\tau : [\![\Gamma]\!] \triangleright_\tau [\![\Gamma']\!] \qquad [\![\Gamma, \alpha_i : T^\perp]\!] \triangleq [\![\Gamma]\!], T^\perp$$

$$\Upsilon \triangleright_{\mathsf{c}} T \triangleq \forall Y <: \Upsilon.\ Y \to \perp$$
$$\Upsilon \triangleright_{\mathsf{t}} T \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{e}} T) \to \perp$$
$$\Upsilon \triangleright_{\mathsf{e}} T \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{V}} T) \to \perp$$

$$\Upsilon \triangleright_{\mathsf{V}} X \triangleq X$$
$$\Upsilon \triangleright_{\mathsf{V}} T \to U \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_{\mathsf{V}} T) \to (Y \triangleright_{\mathsf{e}} U) \to \perp$$
$$\Upsilon \blacktriangleright T \triangleq \Upsilon \triangleright_{\mathsf{V}} T$$
$$\Upsilon \blacktriangleright T^\perp \triangleq \Upsilon \triangleright_{\mathsf{e}} T$$

(b) Translation of types and judgments

Figure 18. Call-by-value continuation-and-environment-passing style translation

H 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 for the call-by-value $\bar{\lambda}\mu\tilde{\mu}$-calculus with explicit environments. Its syntax is given by:

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

while the reduction rules are given by:

(Catch)	$\langle \mu\alpha_i.c \parallel e \rangle \tau$	$\to$	$c[\alpha_n/\alpha_i]\tau[\alpha_n := e]$	with $\lvert\tau\rvert = n$
(Let)	$\langle V \parallel \tilde{\mu} x_i.c \rangle \tau$	$\to$	$c[x_n/x_i]\tau[x_n := V]$	with $\lvert\tau\rvert = n$
(Lookup$_x$)	$\langle V \parallel \alpha_n \rangle \tau$	$\to$	$\langle V \parallel \tau(n) \rangle \tau$	
(Lookup$_\alpha$)	$\langle x_n \parallel E \rangle \tau$	$\to$	$\langle \tau(n) \parallel E \rangle \tau$	
(Beta)	$\langle \lambda x_i.t \parallel u \cdot e \rangle \tau$	$\to$	$\left\langle u \,\middle\|\, \tilde{\mu} x_i.\langle t \parallel e \rangle \right\rangle \tau$	

The main specificity of the translations is that since only values can be stored in environments, we define $\Upsilon \blacktriangleright T = \Upsilon \triangleright_{\mathsf{V}} T$. The translations of types and judgments, 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 operational semantics. Namely, since terms at level t are first analyzed, then context at level e and finally values, the translation follows the same hierarchy. Again, we have:

Theorem H.1. *The translation is well-typed, i.e.:*

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