A calculus of expandable stores
Hugo Herbelin, Étienne Miquey

To cite this version:

HAL Id: hal-02557823
https://hal.archives-ouvertes.fr/hal-02557823v2
Submitted on 29 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 $\lambda$-calculus is an evaluation strategy that lazily evaluates arguments only if needed, and if so, shares computations across all places where it is needed. To implement this evaluation strategy, abstract machines require some form of global environment. While abstract machines usually lead to a better understanding of the flow of control during the execution, 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 $F_\Upsilon$, a generic calculus to define the target of such translations. In particular, $F_\Upsilon$ 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 $F_\Upsilon$ 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 $\lambda$-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 $F_\Upsilon$, 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 Bruijn presentation used in this paper formally addresses this issue by avoiding names altogether. Most importantly, this paper shows that a de Bruijn 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 F1 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) \( \lambda \)-calculus that uses a single global environment [2, 3]. A state of this machine is made of three components:

- a code \( \tilde{t} \) for a term \( t \) which is not considered up to \( \alpha \)-conversion\(^1\),
- a stack \( \pi \) which contains the arguments of the current code, that is to say a stack of codes,
- a global environment \( \tau \), which is a list storing the (delayed) substitutions generated by the redexes encountered so far.

Formally, this corresponds to the following syntax:

\[
\begin{align*}
\text{Terms} & : t, u ::= x \mid tu \mid \lambda x.t \\
\text{Stacks} & : \pi ::= \epsilon \mid \tilde{t} \cdot \pi \\
\text{Environments} & : \tau ::= \epsilon \mid \tau[x := t]
\end{align*}
\]

The machine is given in Figure 1, where we follow the notations of Accattoli and Barras [3] to denote reduction rules: we write \( \rightarrow_{\beta} \) for the \( \beta \)-reduction rule, \( \rightarrow_{s} \) for the rule which somehow performs a substitution, and \( \rightarrow_{c} \) for the commutative transition. By considering invariants of the machine, one can easily prove that the Milner Abstract Machine soundly implements the call-by-name evaluation strategy for the \( \lambda \)-calculus [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 \( \alpha \)-renaming process. Executing twice a term binding a variable \( x \) in different contexts (say \( \lambda x.t \) and \( \lambda x.t \)) could indeed result in linking two different terms to the same name in the environment. This is avoided by asking in the \( \rightarrow_{s} \) rule that if \( x \) is linked to some term \( t \) in the environment \( \tau \), accessing \( x \) results in executing a code \( \tilde{T}^{x} \) that is \( \alpha \)-equivalent to \( t \) and such that any bound name in \( \tilde{T}^{x} \) is fresh with respect to every other names in the term, in the current stack \( \pi \) and in the environment \( \tau \). In the sequel, we will explicitly use de Bruijn levels to handle this kind of issues.

On the contrary, most of the abstract machines of the literature implementing the call-by-name or call-by-value evaluation strategies for the \( \lambda \)-calculus 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 \( \alpha \)-conversion for terms, which stand for the equivalence class of raw syntactic codes up to \( \alpha \)-conversion.
A calculus of expandable stores

Hugo Herbelin and Étienne Miquey

The call-by-need evaluation strategy of the \( \lambda \)-calculus evaluates arguments of functions only when needed, and, when needed, shares their evaluations across all places where the argument is required. Therefore, abstract machines implementing the call-by-need evaluation have to allow for some kind of global environment in order to share computations [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 \( \tau_i[x := t] \tau_2 \): the machine momentarily focuses on the evaluation of \( t \) in \( \tau_i \) while saving the current stack together with the rest of the environment \( \tau_2 \) and the variable \( x \) on the dump. Then, if this computation eventually produces a value \( v \) in an environment \( \tau'_1 \), the machine goes back to the former computation within the updated environment \( \tau'_1[x := v] \tau_2 \). The machine is given Figure 2, in which environments \( r \) and stacks \( \pi \) are defined as in the MAM, and where dumps are given by the following grammar:

\[
D ::= \epsilon | (x, \pi, r) :: D
\]

Alternatively, one could also extend the syntax of stacks in order to add a constructor \( h(x, r) \cdot \pi \) that holds the entries that would otherwise go to the dump. The modified reduction rules could then be given by:

\[
x \cdot \pi \cdot r[x := \tilde{t}] r' \rightarrow c_1 \tilde{t} \cdot \tilde{u} \cdot \pi \cdot r \cdot D
\]

\[
\tilde{u} \rightarrow \pi \cdot r \cdot D \rightarrow c_1 \tilde{t} \cdot \tilde{u} \cdot \pi \cdot r \cdot D
\]

\[
x \cdot \pi \cdot r[x := \tilde{t}] r' \rightarrow D \rightarrow c_2 \tilde{t} \cdot h(x, r') \cdot \pi \cdot r
\]

\[
\tilde{u} \cdot h(x, r') \cdot \pi \cdot r \rightarrow s \tilde{u} \cdot \tilde{v} \cdot \pi \cdot r[x := \tilde{v}] r' \cdot D
\]

This is for instance the approach chosen in Sestoft’s machine for lazy evaluation [33] or in Accattoli, Barendbaum 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, Barendbaum and Mazza’s paper [2].

1.3 The \( \lambda \mu \bar{\mu} \)-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 \( \lambda \mu \bar{\mu} \)-calculus extended with a global environment [9]. The syntax of the usual \( \lambda \mu \bar{\mu} \)-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 \( j x.c \) can be read as a context \( \lambda x \cdot j \) in \( c \). As for terms, the \( \mu \) operator comes from Parigot’s \( \lambda \mu \)-calculus [30]: \( \mu A \) binds a context to a context variable \( a \) in the same way \( j x \) 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 \( \alpha \)-conversion in order to preserve the uniqueness of names in the environment. It is easy to see that on its intuitionistic fragment (that is without the classical control \( \mu a \)), this calculus behaves exactly as the MAM\(^2\). As per [9], we restrict the

\(^2\)The reader unfamiliar with the \( \lambda \mu \bar{\mu} \)-calculus might be puzzled by the absence of a syntactic construction for the application of proof terms. Intuitively, the usual application \( tu \) of the \( \lambda \)-calculus is replaced by the application of the proof \( t \) to a stack of the shape \( u : E \). The usual application can be recovered through the following shorthand: \( tu \equiv \mu a.(r \parallel u \cdot a). \) It is then an easy exercise to show that the MAM can be simulated within the \( \lambda \mu \bar{\mu} \)-calculus with environments, see Appendix A.

---

**Figure 1. MAM vs KAM**

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

- **Terms**: \( t, u ::= x \mid tu \mid \lambda x.t \)
- **Stacks**: \( S ::= \epsilon \mid (E, S) \)
- **Environments**: \( E ::= \epsilon \mid E[x := (t, E')] \)

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 \( \alpha \)-renaming process useless. While this design with a local environment presents some benefits over machines with global ones (among other things in terms of complexity [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 \( \lambda \)-calculus evaluates arguments of functions only when needed, and, when needed, shares their evaluations across all places where the argument is required. Therefore, abstract machines implementing the call-by-need evaluation have to allow for some kind of global environment in order to share computations [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 \( \tau_i[x := t] \tau_2 \): the machine momentarily focuses on the evaluation of \( t \) in \( \tau_i \) while saving the current stack together with the rest of the environment \( \tau_2 \) and the variable \( x \) on the dump. Then, if this computation eventually produces a value \( v \) in an environment \( \tau'_1 \), the machine goes back to the former computation within the updated environment \( \tau'_1[x := v] \tau_2 \). The machine is given Figure 2, in which environments \( r \) and stacks \( \pi \) are defined as in the MAM, and where dumps are given by the following grammar:

\[
D ::= \epsilon | (x, \pi, r) :: D
\]

Alternatively, one could also extend the syntax of stacks in order to add a constructor \( h(x, r) \cdot \pi \) that holds the entries that would otherwise go to the dump. The modified reduction rules could then be given by:

\[
x \cdot \pi \cdot r[x := \tilde{t}] r' \rightarrow c_1 \tilde{t} \cdot \tilde{u} \cdot \pi \cdot r \cdot D
\]

\[
\tilde{u} \rightarrow \pi \cdot r \cdot D \rightarrow c_1 \tilde{t} \cdot \tilde{u} \cdot \pi \cdot r \cdot D
\]

\[
x \cdot \pi \cdot r[x := \tilde{t}] r' \rightarrow D \rightarrow c_2 \tilde{t} \cdot h(x, r') \cdot \pi \cdot r
\]

\[
\tilde{u} \cdot h(x, r') \cdot \pi \cdot r \rightarrow s \tilde{u} \cdot \tilde{v} \cdot \pi \cdot r[x := \tilde{v}] r' \cdot D
\]
Although the MAD arguably provides us with the easiest typing rules, they will be easy to deduce from the presentation of a lazy abstract machine, it does not directly address the problem of soundly defining the 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 later 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 λµΓ-calculus, they finally obtain a small-step sequent calculus, the $\overline{\lambda[\mu\nu^{\star}]}$-calculus, from which they get an untyped CPS translation almost for free\(^3\). The $\overline{\lambda[\mu\nu^{\star}]}$-calculus can be understood as a refinement of the $\overline{\lambda\mu}$-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 $\tilde{\mu}[x].(x \parallel F)^{\tau'}$, 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:

$$(\mu \alpha.c \parallel \tilde{\mu}x.c)^{\tau} \rightarrow c^{\tau'}[x := \mu \alpha.c]$$

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 $(x \parallel F)^{\tau}[x := \mu \alpha.c]^{\tau'}$, the binding is opened and the term is evaluated in front of the context written $\tilde{\mu}[x].(x \parallel F)^{\tau'}$:

$$(x \parallel F)^{\tau}[x := \mu \alpha.c]^{\tau'} \rightarrow (\mu \alpha.c \parallel \tilde{\mu}[x].(x \parallel F)^{\tau'})^{\tau}$$

The reader can think of the previous rule as the “defrosting” operation of the frozen term $\mu \alpha.c$; this term is evaluated in the prefix of the environment $\tau$ which predates it, in front of the context $\tilde{\mu}[x]$. $(x \parallel F)^{\tau'}$ where the $\tilde{\mu}[x]$ binder is waiting for a value. This context keeps trace of the part of the environment $\tau'$ that was originally located after the binding $[x := \ldots]$. This way, if a value $V$ is indeed furnished for the binder $\tilde{\mu}[x]$, the original command $(x \parallel F)$ is evaluated in the updated full environment:

$$(V \parallel \tilde{\mu}[x].(x \parallel F)^{\tau'})^{\tau} \rightarrow (V \parallel F)^{\tau}[x := V]^{\tau'}$$

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 $\overline{\lambda[\mu\nu^{\star}]}$-calculus are close to the ones of the MAD\(^5\).

\(^3\)See Appendix B.1 for the original definition of the $\overline{\lambda[\mu\nu^{\star}]}$-calculus and the corresponding untyped CPS.

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

\(^5\)In fact, it is an easy exercise to simulate the MAD within the $\overline{\lambda[\mu\nu^{\star}]}$-calculus. We outline this construction in Appendix A.
A calculus of expandable stores
Hugo Herbelin and Étienne Miquey

Strong values
\[ v ::= k \mid \lambda x_i.t \]

Weak values
\[ V ::= v \mid x_i \]

Terms
\[ t, u ::= V \mid \mu a_i.c \]

Stores
\[ \tau ::= e \mid \tau[x_i := t] \mid \tau[a_i := E] \]

Commands
\[ c ::= \tau \parallel e \]

Closures
\[ l ::= c \tau \]

Forcing contexts
\[ F ::= k \mid t \cdot E \]

Catchable contexts
\[ E ::= F \mid a_i \mid \mu [x_i].(x_i \parallel F) \tau \]

Evaluation contexts
\[ e ::= E \mid \mu x_i.c \]

\begin{align*}
\text{(Let)} & \quad \langle t \parallel \mu x_i.c \rangle \tau \quad \rightarrow^\dagger \quad c[x_n/x_i]_{\tau[x_n := t]} \\
\text{(Catch)} & \quad \langle \mu a_i.c \parallel E \rangle \tau \quad \rightarrow^\dagger \quad c[a_i/a_i]_{\tau[\alpha_n := E]} \\
\text{(Lookup}_a\rangle & \quad \langle V \parallel a_n \rangle \tau \quad \rightarrow \quad \langle V \parallel \tau(n) \rangle \tau \\
\text{(Lookup}_n\rangle & \quad \langle x_n \parallel F \rangle \tau_0[x_n := t] \tau_1 \rightarrow \quad \langle t \parallel \mu x_n.(x_n \parallel F) \tau_1 \rangle \tau_0 \\
\text{(Restore)} & \quad \langle V \parallel \mu [x_i].(x_i \parallel F) \tau' \rangle \tau \rightarrow^\dagger \quad \langle V \parallel \tau F \tau[x_n := V] \tau \rangle \tau \\
\text{(Beta)} & \quad \langle \lambda x_i.t \parallel u \cdot E \rangle \tau \rightarrow^\dagger \quad \langle u \parallel \mu x_n.(\ell[x_n/x_i] \parallel E) \tau \rangle \tau
\end{align*}

\(\dagger\) with \(|\tau| = n\)

As the environment can be dynamically extended during the execution, the location of a term in the environment and the corresponding pointer are likely to evolve (monotonically). Therefore, we need to be able to update de Bruijn levels within terms (contexts, etc.). To this end, we define the lifted term \(\uparrow^n t\) as the term \(t\) where the free variables \(x_j\) (resp. \(a_j\)) with \(j \geq i\) have been replaced by \(x_{j+n}\) (resp. \(a_{j+n}\)). 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 \(n+1\)th element of the environment.

**Figure 4.** The \(\widetilde{\lambda}_{[t \parallel r \parallel \tau]}\)-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 \(\mu a\) 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 \(\widetilde{\lambda}_{[t \parallel r \parallel \tau]}\)-calculus deeply relies on the assumption that names of variable are unique and thus on the possibility of performing \(\alpha\)-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. In turn, we will use de Bruijn levels for variables (and co-variables) that are bound in the environment. Just as de Bruijn indices are pointers to the correct binder, de Bruijn levels are pointers to the correct cell of the environment. We use the 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 \(a_i\), where, again, \(a\) 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 \(\uparrow^n t\) as the term \(t\) where the free variables \(x_j\) (resp. \(a_j\)) with \(j \geq i\) have been replaced by \(x_{j+n}\) (resp. \(a_{j+n}\)). 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 \(n+1\)th element of the environment.

**Theorem 1.1** (Subject reduction). If \(\Gamma \vdash t \tau\) and \(t \tau \rightarrow c't'\) then \(\Gamma \vdash c'\tau'\).

\(\dagger\)See Appendix B.2 for more details on problems related to \(\alpha\)-conversion
\(\dagger\)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.
we illustrate each of them with the translation of the sequent\(^9\) \(a : A, \alpha : A^\|, b : B \tau e : C\).

**Step 1 – Continuation-passing style.** In a first approximation, let us look only at the continuation-passing style part of the translation of a \(\cal A_{\{l\tau r\}^*}\) sequent. As emphasized in [5, 26], there are 6 different levels of control in the \(\cal A_{\{l\tau r\}^*}\) calculus (corresponding to the 6 nested syntactic categories), leading to 6 mutually defined levels of interpretation. We define \(\lbrack A \rightarrow B\rbrack_v\) for strong values as \(\lbrack A\rbrack_v \rightarrow \lbrack B\rbrack_v\), we define \(\lbrack A\rbrack_F\) for forcing contexts as \(\lbrack A\rbrack_v\) for weak values as \(\neg \lbrack A\rbrack_F = \neg \lbrack A\rbrack_v\), and so on until \(\lbrack A\rbrack_{v^5} = \neg \lbrack A\rbrack_{v^4}\) (where we use the notations \(v^1 A \equiv A \rightarrow \bot\) and \(v^5 A \equiv \neg A\)).

As observed in the realizability interpretation [26], hypotheses from a context \(\Gamma\) of the form \(a : A^\|\) are to be translated as \(\lbrack A\rbrack_{v^5}\) while hypotheses of the form \(x : A\) are to be translated as \(\lbrack A\rbrack_e = 4^1 \lbrack A\rbrack_{v^5}\). Up to this point, if we denote this translation of \(\Gamma\) by \(\lbrack\Gamma\rbrack\), in the particular case of \(\Gamma \vdash A\) the translation is \(\lbrack\Gamma\rbrack \vdash \lbrack A\rbrack\), and similarly for other levels.

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

\[
\begin{align*}
\llbracket a : A, \alpha : A^\|, b : B \tau e : C \rrbracket_v & = \llbracket a : A\rbrack_v, \llbracket \alpha : A^\|\rbrack_v, \llbracket b : B\rbrack_v, \llbracket e : C\rbrack_v \\
& = \llbracket a : A\rbrack_{v^5}, \llbracket \alpha : A^\|\rbrack_{v^5}, \llbracket b : B\rbrack_{v^5}, \llbracket e : C\rbrack_{v^5}
\end{align*}
\]

**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 \(\Gamma \vdash A\) is not anymore a sequent \(\lbrack\Gamma\rbrack \vdash \lbrack A\rbrack\), but instead a sequent roughly of the form \(\lbrack\Gamma\rbrack \vdash \lbrack A\rbrack\), with actually \(\lbrack\Gamma\rbrack\) being passed around not only at the top-level of \(\lbrack A\rbrack\), but also each time a negation is used. We write this sequent \(\lbrack\Gamma\rbrack \vdash A\) where \(\vdash\) is defined by induction on \(t\) and \(A\):

\[
\begin{align*}
\llbracket\Gamma\rbrack \vdash A & = \llbracket\Gamma\rbrack \Rightarrow \llbracket\Gamma\rbrack \vdash E \Rightarrow \bot \\
& = \llbracket\Gamma\rbrack \Rightarrow \llbracket\Gamma\rbrack \Rightarrow \bot \Rightarrow \bot \Rightarrow \cdots
\end{align*}
\]

\(^9\)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 $\Gamma$ 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 $a : A, \alpha : A^4, b : B \vdash e : C$ is:

$$\begin{align*}
[a : A, \alpha : A^4, b : B] &\vdash [e]_e : [a : A, \alpha : A^4, b : B] \triangleright e C \\
[a : A, \alpha : A^4, b : B] &\vdash [e]_e : [\alpha : A, \alpha : A^4, b : B] \triangleright e \Gamma \\
[a : A, \alpha : A^4, b : B] &\vdash [\alpha : A, \alpha : A^4, b : B] \triangleright e \Gamma C \\
[a : A, \alpha : A^4, b : B] &\vdash [\alpha : A, \alpha : A^4, b : B] \triangleright e \Gamma C
\end{align*}$$

where:

$$\begin{align*}
[a : A, \alpha : A^4, b : B] &= a : A, \alpha : A^4 \\
[a : A, \alpha : A^4] &= \alpha : A, \alpha : A^4 \\
\end{align*}$$

**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 a system type an adaptation of System $F_c$ of [7] extended with stores, defined as lists of assignations $[x := t]$. Store types, denoted by $\Gamma$, 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 $\Gamma' <: \Gamma$ to express that $\Gamma'$ is an extension of $\Gamma$. This corresponds to the following refinement of the definition of $[\Gamma] \triangleright e A$:

$$\begin{align*}
[\Gamma] \triangleright e A &= \forall Y <: [\Gamma], Y \rightarrow (Y \triangleright e A) \rightarrow \perp \\
&= \forall Y <: [\Gamma], Y \rightarrow (\forall Y' <: Y, Y' \rightarrow Y' \triangleright e A \perp \perp) \rightarrow \perp \\
&= \ldots
\end{align*}$$

Such a quantification is reminiscent of Kripke forcing [16]: thinking of store types $\Gamma$ as worlds, the accessible worlds from $\Gamma$ are precisely all the possible $\Gamma' <: \Gamma$. 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:

$$\begin{align*}
Y \triangleright e T &\rightarrow U \triangleq \forall Y <: Y, Y \rightarrow (Y \triangleright e T) \rightarrow (Y \triangleright e U) \rightarrow \perp \\
\vdash \alpha : A \Rightarrow B \triangleq \forall \alpha' \geq \alpha, \alpha' \vdash A \Rightarrow \alpha' \vdash B
\end{align*}$$

**Example 2.3 (Translation, step 3).** The translation, now taking into account store extensions, of $a : A, \alpha : A^4, b : C$ becomes:

$$\begin{align*}
[a : A, \alpha : A^4, b : B \vdash e : C] &= \vdash [e]_e : [a : A, \alpha : A^4, b : B] \triangleright e C \\
[a : A, \alpha : A^4, b : B] &= \vdash [e]_e : \forall Y <: [a : A, \alpha : A^4, b : B], Y \rightarrow (Y \triangleright e C) \rightarrow \perp \\
\vdash [a : A, \alpha : A^4, b : B] &= \vdash [a : A, \alpha : A^4] \triangleright e \Gamma C
\end{align*}$$

where:

$$\begin{align*}
[a : A, \alpha : A^4, b : B] &= [a : A, \alpha : A^4] \triangleright e \Gamma C \\
[a : A, \alpha : A^4] &= [a : A, \alpha : A^4] \triangleright e \Gamma C \\
[a : A, \alpha : A^4] &= [a : A, \alpha : A^4] \triangleright e \Gamma C
\end{align*}$$

**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. $A, A^4, \ldots$ instead of $a : A, \alpha : A^4, \ldots$).

Interestingly, this requires giving computational content to the subtyping relation $\Gamma' <: \Gamma$ through explicit coercions $\sigma : \Gamma' <: \Gamma$ mapping each type in $\Gamma$ to the corresponding type in $\Gamma'$. In other words, considering a store $\tau : \Gamma$ extending a store $\tau' : \Gamma'$, a coercion $\sigma : \Gamma' <: \Gamma$ indicates where each element of $\tau$ can be found in $\tau'$.

**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 $t$, which is itself defined as the same transformation applied to the translation at level $t+1$ and so on. Each step precisely consists in refining this transformation. Formally, if $\mathcal{F}$ is a (meta) function taking a store type $\Gamma$ and returning a type, let us define:

$$\square \mathcal{F} \triangleq \Gamma \rightarrow \forall Y' <: Y, Y' \rightarrow (\mathcal{F} Y') \rightarrow \perp$$

which is also a function mapping store types $\Gamma$ to types in the target of the translation. Then, writing $\triangleright e A$ for the function associating to any store type $\Gamma$ the type $e : A$ (for $i$ a level among $e, t, \ldots$, the definition of the translation can be simply expressed by:

$$\triangleright e A = \square (\triangleright e \Gamma A) = \square (\triangleright e \Gamma A) = \ldots$$

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

- at step 1, $\square \mathcal{F}$ is $\Gamma \rightarrow (\mathcal{F} \Gamma) \rightarrow \perp$,
- at step 2, $\square \mathcal{F}$ is $\Gamma \rightarrow \forall Y <: \Gamma, Y' \rightarrow (\mathcal{F} Y') \rightarrow \perp$,
- at step 3, $\square \mathcal{F}$ is $\Gamma \rightarrow \forall Y' <: \Gamma, Y' \rightarrow (\mathcal{F} Y') \rightarrow \perp$.

3 $\mathcal{F}_c$ 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 $\mathcal{F}_c$. Each of the translations that we
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-calculi10 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 $A, A^2, B$ 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 $F_t$. 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 $F_t$. As we explained, we will focus on the case of a simply typed source calculus, whose types are defined by:

```
Source types     $A ::= X | A \to B \quad F ::= A | A^4$
```

As explained above, a store $\tau$ is simply given as a list whose type, written $Y$, is defined as the list of the types of the elements $\tau$ provides. Observe that if we consider the concatenation of such two stores $\tau_0; \tau_1$ of type $Y_0; Y_1$, thinking of them as substitutions, $\tau_1$ provides elements for $Y_1$ only if it is given with a prefix providing $Y_0$ (see also the Step 2 described earlier). We denote its type with $Y_0 \triangleright \tau_1$, expressing the fact that it is well-typed only when concatenated with a store of type $\emptyset \triangleright \tau_0$ (the type of a “self-sufficient” store, which we will abbreviate by $Y_0$). Obviously, such a store should also be compatible with any extension of $Y_0$, which justifies the introduction of the constructor $\uparrow_\tau^\tau \sigma$ where $\sigma$ is of type $Y_0^\tau < Y_0$ and the resulting store is of type $Y_0^\tau \triangleright \tau_1$.

Coercions are simply defined through the natural structure witnessing list inclusion: $\sigma^+$ (resp. $\uplus \sigma$) witnesses the fact that $Y, F$ (resp. $Y$) is included in $Y', F$ provided that $\sigma$ witnesses that $Y$ is included in $Y'$. Having in mind that list inclusion corresponds to a subtyping relation for stores, we denote11 it by $Y' < Y$. We denote by $\sigma \circ \sigma$ the composition of two coercions, and by $\sigma^{+1}$ (resp. $\uplus \uplus \sigma$) the twin of the previous coercions witnessing the addition of a list of type $Y$.

Finally, System $F_t$ terms are defined as the usual $\lambda$-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

---

10In fact, variants of the $\lambda\mu\nu$-calculus with explicit stores, but which would naturally define an Intermediate Representation for the corresponding $\lambda$-calculi in the compilation chain.

11This may read “$Y$ is more precise than $Y$ 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:

\[
\text{Store type } Y ::= Y | 0 | F | \tau Y
\]

\[
\text{Stores } \tau ::= \delta \mid [t] \mid [t] \tau \mid [t] \tau
\]

\[
\text{Coercions } \sigma ::= s | \varepsilon | \sigma' | [\sigma] \sigma
\]

\[
\text{Contexts } \Gamma ::= \vdash | \Gamma, s : \tau Y < Y | \Gamma, \delta : Y \Rightarrow \tau Y < \Gamma, x : T
\]

\[
\text{Types } T ::= X | T \rightarrow U | \forall Y.T | Y < Y \rightarrow T | Y \Rightarrow T
\]

\[
\text{Terms } t ::= k | x | \lambda t. t | u \gg \lambda t. t \mid t \sigma | \lambda \delta t. t \mid t \tau
\]

As for the typing rules, there are given in Figure 6 where \( S \) 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 \( \equiv_T \) is the congruent-transitive-symmetric-reflexive closure of the reduction \( \rightarrow_T \) (defined in the next section) and where \( \Upsilon \) is of the shape 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 \( \tau \) can be lifted to match extensions of \( \Upsilon \), for our system to be sound, we need to assume that terms in the store can be lifted as well. Formally, we require that \( \Upsilon \rightarrow \Upsilon \) is of the shape\(^{13}\):

\[
(\text{Eq}_\rightarrow) \quad \Upsilon \rightarrow \Upsilon \equiv \forall Y < \Upsilon. \forall F(Y, F)
\]

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

\[
\text{split } \tau \text{ at } n \text{ along } \sigma : Y < \Upsilon \text{ as } (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1) \text{ in } t
\]

to be well-typed, we need to ensure that \( \Upsilon \) is of the shape \( (\Upsilon_0, \Upsilon_1, \Upsilon) \) where we abuse the notation \( [\Upsilon_0] \) 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 \( \Box \forall Y < : \Upsilon. \forall F(Y, F) \rightarrow \bot \) for any \( Y \). We have:

**Lemma 3.1.** Assume that for any \( Y, \forall F(Y, F) \) is a type. The following rules are admissible:

\[
\frac{\Gamma, s : Y < : \Upsilon, \delta : \emptyset \Rightarrow \tau Y, x : \forall F(Y, F) \rightarrow \bot}{\Gamma \vdash \lambda s \delta x. t : \forall F(Y, F)} \quad (\Box_1)
\]

\[
\frac{\Gamma \vdash t : \Box \forall F(Y, F) \quad \Gamma \vdash \sigma : \Upsilon < \Upsilon \quad \Gamma \vdash u : \forall F(Y, F)}{\Gamma \vdash t \sigma u : \bot} \quad (\Box_2)
\]

As we shall see in the next sections, we will mainly use these "packed" abstractions and applications to define the different translations (observe that \( \Box F Y \) satisfies (Eq_\rightarrow)). 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 \( F_1 \). Actually, we could have presented \( F_1 \) 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 \( F_1 \). If most of them are straightforward, some special care has to be given to define the reduction of terms of the shape \( \tau \rightarrow n \) along \( \sigma : Y < : \Upsilon \) as \( ... \rightarrow t \). To reduce such a term, we will need to use the coercion \( \sigma \) to split the store \( \tau \) correctly. In particular, we need \( \sigma \) (and \( \tau \)) to be in normal form, that is that it contains neither variables nor compositions of coercions:

**Normal**

\[
\tau_n ::= [t] | \tau_n [t]
\]

**Forms**

\[
\Upsilon_n ::= \emptyset | \Upsilon_n : F
\]

\[
\sigma_n ::= \varepsilon | \sigma_n' | \Box \sigma_n
\]

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 \( \Gamma \vdash \sigma : Y < : [\Upsilon_0], A; [\Gamma_1] \) where \( \sigma \) is a coercion in normal form then there exists \( \Upsilon_0, \Upsilon_1, \sigma_0, \sigma_1 \) such that:

1. \( \Gamma \vdash \sigma_0 : \Upsilon_0 < : [\Upsilon_0] \)
2. \( \Gamma \vdash \sigma_1 : \Upsilon_1 < : [\Upsilon_1] \)
3. \( \Upsilon = \Upsilon_0; A; \Upsilon_1 \)
4. \( \sigma = \sigma_0 \delta \sigma_1 \)

where \( \sigma_0; \sigma_1 \) is the coercion \( \sigma_1 \) where \( \sigma_0 \) replaces \( \varepsilon \).

Alternatively, a coercion \( \sigma : Y < : \Upsilon \) in normal form can actually be identified with a partial monotone function \( [\sigma] \) from \( [0, \Upsilon \Rightarrow T] \) to \( [0, \Upsilon \Rightarrow T] \) which intuitively maps the index of every type in \( \Upsilon \) to its corresponding index in \( Y \) (see Appendix D.1).

**Example 3.3.** As an example, for any types \( T, U, T_0, T_1 \), if we write \( \sigma \) to denote the coercion \( [\Box \forall F(Y, F)] \) which is in normal form, we have:

- \( \vdash [\Box \forall F(Y, F)] : T_0, U, T_1 : T, U \rightarrow T, U \)
- \( \text{dom}(\sigma) = 2, \text{codom}(\sigma) = 4 \)
- \( [\sigma]_N = [0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 4] \)

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

Similarly, it is easy to see that if a store \( \tau \) is in normal form and is of type \( [\Upsilon_0], A; [\Gamma_1] \), then we can divide it as expected into \( \tau = \tau_0 | u ; \tau_1 \). This justifies the reduction rule for split:

\[
\text{split } \tau_0[u]; \tau_1 \text{ at } n \text{ along } \sigma : \Upsilon_0, A, \Upsilon_1 < : [\Upsilon_0], A; [\Gamma_1] \text{ as } (\Upsilon_0, s_0, \delta_0), x, (\Upsilon_1, s_1, \delta_1) \text{ in } t \rightarrow t[\Upsilon_0/\Upsilon_0]?\sigma_0/s_0|\tau_0/\delta_0]|u/x|\Upsilon_1/\Upsilon_1]t_1/\delta_1]_1/s_1\]

\[\text{as } (\Upsilon_0, s_0, \delta_0), x, (\Upsilon_1, s_1, \delta_1) \text{ in } t \rightarrow t[\Upsilon_0/\Upsilon_0]?\sigma_0/s_0|\tau_0/\delta_0]|u/x|\Upsilon_1/\Upsilon_1]t_1/\delta_1]_1/s_1\]
where n = |Γ₀|, |σ₀| = |Γ₀| and Y₀, Y₁, σ₀, σ₁ are as in Lemma 3.2 (i.e. |Γ₀| = |σ|₀(n), σ₀ : Y₀ : Γ₀, 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:
\[
\begin{align*}
(σ^+ + 1)σ_0 & \rightarrow σ | (σ_0 \circ σ_0) \\
(σ^+ + 1)σ_0 & \rightarrow (σ_1 \circ σ_0) \\
(σ_1 \circ σ_0) & \rightarrow (σ_1 \circ σ_0) \\
(σ_1 \circ σ_0) & \rightarrow (σ_1 \circ σ_0) \\
(Γ, σ) & \rightarrow (Γ, σ) \\
(Γ, σ) & \rightarrow (Γ, σ) \\
(Γ, σ) & \rightarrow (Γ, σ) \\
(Γ, σ) & \rightarrow (Γ, σ) \\
\end{align*}
\]

where \( Γ \overset{σ}{\rightarrow} \lambda Y.s.t \ Y s o σ. \)

b) Usual β-reduction steps for the different abstractions:
\[
(λx.t) u \rightarrow t[u/x] \\
(λy.t) Y \rightarrow t[Y/x] \\
(λδ.t) \rightarrow t[τ/δ]
\]

c) Contextual rules for store types in stores and coercions:
\[
\begin{align*}
U_σ[Y] & \overset{τ \rightarrow τ'}{\rightarrow}_σ U_σ[Y'] \\
U_τ[Y] & \overset{τ \rightarrow τ'}{\rightarrow}_τ U_τ[Y']
\end{align*}
\]

where \( U_σ[.] := \sigma; U_τ[.] := τ \)

d) Contextual rules for stores and coercions in split:
\[
\begin{align*}
C[σ] & \overset{σ \rightarrow σ'}{\rightarrow} C[σ'] \\
D[τ] & \overset{τ \rightarrow τ'}{\rightarrow} D[τ']
\end{align*}
\]

where contexts are defined by:
\[
\begin{align*}
C[.] & := \text{split}[.] \text{ at } n \text{ along } σ : Y' < Y \text{ as } ... \text{ in } t \\
D[.] & := \text{split } r \text{ at } n \text{ along } σ : Y' < Y \text{ as } ... \text{ in } t
\end{align*}
\]

**Theorem 3.4** (Subject reduction). If \( Γ \vdash t : T \) and \( t \rightarrow t' \) then \( Γ \vdash t' : T \).

*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 use \( F₁ \) 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 \( F₁ \) (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 \( τ + 1 \)). 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 \( F_{c<} \).

In turn, it is clear that various dependent type theories are expressive enough to define \( F₁ \). As an example, in Figure 7 we give an overview of how \( F₁ \) can be expressed in Coq through a shallow embedding. We give more details in Appendix E.\(^{13}\)

### 4 Implementation for simply typed calculi

We are now equipped to define typed continuation-and-environment-passing style translations to system \( F₁ \). These translations exactly follow the intuitions outlined in Section 2.1. We first focus on the case of the (call-by-need) \( \tilde{\Lambda}_{[\text{tor}]} \)-calculus, and then illustrate the generality of our method by giving a translation for the call-by-name \( \tilde{\Lambda}_{\mu} \)-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.

---

\(^{13}\)The Coq development illustrating these ideas is available at: https://gitlab.com/emiquey/fupsilon.
4.1 A typed call-by-need translation for the \(\lambda[[\text{ut}]]\)-calculus

4.1.1 Translation of terms

We can now take advantage of the features of system F\(_1\) to type Ariola et al.’s untyped translation for the \(\lambda[[\text{ut}]]\)-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 \(\tau[t]; t'\), \(t\) has its levels coherent with its prefix \(\tau\), and ignores its suffix \(t'\). In terms of types, if \(\tau[t]\) is of type \(Y, A\) then \(t\) will be of a type \(Y, A\).
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 \([x: Y]\) and \([\bar{\mu}[x: Y, x: F] t:\tau']\). In the untyped translation (see Appendix B.1), we have:

\[
\begin{align*}
\llbracket \mu[x: Y, x: F] t:\tau' \rrbracket_F & \triangleq \llbracket \mu[x: Y, x: F] t:\tau' \rrbracket_{\text{F}_{\text{ut}}} \\
\llbracket x \rrbracket_Y & \triangleq i \\
\llbracket \mu \rrbracket_Y & \triangleq \nu \llbracket \mu \rrbracket_Y
\end{align*}
\]

Let us first focus on the \([\bar{\mu}[x: Y, x: F] t:\tau']\). In the named version, its translation is a term which waits for a store \(\tau\) and a value \(V\), then forms a store looking like \(\tau[x := V]\) and passes it to \(V\) with the continuation \([F]_\nu\). Now, when using de Bruijn levels, the continuation was expecting a store \(\tau_0\) of type \(Y, 0\) that \(\tau\) (the actual store, of type \(Y\)) is extending, hence \(\tau\) comes together with a coercion \(\sigma : Y : Y, 0\) witnessing that \(Y\) is an extension of \(Y, 0\). Let us loosen 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 \(\tau\), but \([F]_\nu\) and \([\tau']_\nu\) have to be updated. In detail, \([\tau']_\nu\) was expecting \(\tau_0[\tau_1]\), thus we update it with \(\sigma^{\tau_0[\tau_1]}\) which witnesses \(\tau[\tau_1] < \tau_0[\tau_1]\). On the other hand, \(F\) was waiting for \(\tau_0[\tau_1]\), \(\tau'\), we thus need to update it with \(\sigma' = (\sigma^{\tau_0[\tau_1]})^{\tau'}\). Finally, we need to give to \(V\) a coercion witnessing the extension of \(\tau\) into \(\tau[\tau_1]\) and \(\tau'\), that is to say \(\llbracket \tau[\tau_1] \rrbracket_Y\). In the end, we obtain the following definition:

\[
\llbracket \bar{\mu}[x: Y, x: F] t:\tau' \rrbracket \triangleq \llbracket \bar{\mu}[x: Y, x: F] t:\tau' \rrbracket_{\text{F}_{\text{ut}}}
\]

As for \([x: Y]\), it is a term waiting for a coercion \(\sigma\) and a store \(\tau\), which it will split at \([\sigma]_{\text{F}_{\text{ut}}}(i)\) as \(\tau_0, \tau, \tau_1\) to execute the term \(t\) with its prefix \(\tau_0\) (with which it is already consistent) and a continuation refining the translation of \(\bar{\mu}[x: Y, x: F] t:\tau'\).

The translation of terms is given in Figure 8, where we assume that for each constant \(k\) of type \(T\) (resp. co-constant \(k\) of type \(T^\perp\)) of the source calculus, we have a constant of
type $\tau$ in the signature of the target language that we also denote by $k$ (resp. $\kappa$ of type $\tau \to \bot$).

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 $\tau$ function. A typing context $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ is then translated into the store type $[\Gamma] = T_1 \ldots T_n$. This allows us to translate a sequent $\vdash t : \tau$, into a judgment $\Gamma \quad \vdash \Gamma \tau t : \tau$. The type $[\Gamma] \tau T$ can be understood as the types of terms translated at level $r$, which are waiting for any store extending $[\Gamma]$ and a continuation at the inferior level (i.e. $E$) for the very same type $\tau$ but interpreted with the extended store type $Y$:

$$[\Gamma] \tau T = \forall Y <: [\Gamma]. Y \to (Y \tau E T) \to \bot$$

It is worth noting that the translation $\tau$ is defined internally in system $F_t$, which allows in particular the recursive definition $Y \tau E T$ to make sense (since $Y$ is at the moment an interleaving of types in $\Gamma$ and second-order variables that are not the images of types in the source calculus). As we already explained, the different levels $e, t, E, F, u$ of translation reflect the dynamics of the (context-free) reduction system of the $\lambda_{\mu\bar{\xi}}$-calculus, that is to say the different syntactic categories which are examined successively during the reduction.

The resulting translation, which is given in Figure 9, is sound, i.e. the provability of a sequent in the source calculus (say $\Gamma \vdash t : \tau$) entails the provability of its translation $([\Gamma] \vdash t : \tau)$ in $F_t$, 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 $\uparrow_t^l$ and $\uparrow_t^*$. The complete proof is given in Appendix F.

4.2 A typed call-by-name translation for the $\lambda_{\mu\bar{\xi}}$-calculus

To emphasize that $F_t$ 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 $\lambda\mu\bar{\xi}$-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:

$$\begin{align*}
\langle t \bar{\langle} \mu x_1. c \rangle \rangle \tau & \rightarrow c[x_n/x_1]_\tau \text{ with } |\tau| = n \\
\langle \mu a_1. c \| E \rangle \tau & \rightarrow c[a_1/a_1]_\tau \text{ with } |\tau| = n \\
(x_n \| E) \tau & \rightarrow (\tau(n) \| E) \tau \\
(V \| a_n) \tau & \rightarrow (V \| \tau(n)) \tau \\
(\lambda x_1. t \| u \cdot E) \tau & \rightarrow (u \bar{\langle} \mu x_1. \langle t \| E \rangle \rangle) \tau
\end{align*}$$

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 $\lambda_{\mu\bar{\xi}}$-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 $\lambda_{\mu\bar{\xi}}$-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 Bruijn 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.

---

10The context-free reduction rules are given in Appendix C.
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 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 ⊢ t : C, through the translation we will thus have t : A, B ⊢ t : C → D. Now, imagine that we dispose of three values V₀, V₁, V₂ respectively of types A, B, C, we can thus construct three closed terms t₀, t₁, t₂ such that, given a continuation, t₁ is going to produce arbitrary computations (and in particular store arbitrarily many terms, let us denote the resulting store by t₁ : U₁) before returning V₁ to its continuation. These terms can thus be assigned the types (A → D) → D, (B → D) → D, (C → D) → D, and the closed term t₀ ≜ t₀((λx.t₁)(λy.t₂u)) can thus be typed by ⊢ t₀ : D. Now, if t₀ is evaluated in an initially empty store, at the moment where uV₂ will be evaluated, the store will be τ₀[x := V₀]τ₁[y := V₁]τ₂ of type U₀, A, U₁, B, U₂.

### 4.3 Using System F as source calculus

In addition to being compatible with different evaluation strategies, Fₜ 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]:

\[ \forall x.F ≜ \forall X : Y → T).Y \triangledown F \quad Y \triangledown X ≜ X Y \]

### 5 Conclusion

**Conclusion** Through the definition of Fₜ, 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 parameteric: 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 $\lambda_{[\text{for}]}$-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 $F_\prec$ [7] and dependently typed calculi (to manipulate sized lists). Yet, the precise logical strength of $F_\prec$ is unclear and is still to determine.

Last but not least, we would like to invest whether the $\square$ can be considered as a modality, and in particular whether we could optimize the translation by using a polarized source calculus, inserting the $\square$ 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, Luis Pinto, Zena Ariola, Paul Downen, Alexis Saurin and Rossen Mikhov for initial discussions on typing the continuation-passing style semantics of call-by-need $\lambda$-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


We then say that two closures $c$ with:

$$
\begin{align*}
\overline{\mu} \star \pi \star \tau & \rightarrow_c \overline{\mu} \star \pi \star \tau \\
\lambda x. \overline{\mu} \star u \star \pi \star \tau & \rightarrow_\mu \overline{\mu} \star \pi \star \tau[x := \overline{\mu}]
\end{align*}
$$

\text{a. Milner Abstract Machine}

\begin{align*}
\langle t \parallel \mu x . c \rangle \tau & \rightarrow c r[x := t] \\
\langle \mu a . c \parallel E \rangle \tau & \rightarrow c r[a := E] \\
\langle V \parallel a \rangle r[a := E] \tau' & \rightarrow \langle V \parallel E \rangle r[a := E] \tau' \\
\langle x \parallel E \rangle r[x := t] \tau'[x := t] & \rightarrow \langle t \parallel E \rangle r[x := t] \tau' \\
\langle \lambda x . t \parallel u \parallel E \rangle \tau & \rightarrow \langle u \parallel \mu x . \langle t \parallel E \rangle \rangle \tau
\end{align*}

\text{b. The $\overline{\lambda}\mu\overline{\mu}$-calculus with global environments}

Figure 11. Milner Abstract Machine and $\overline{\lambda}\mu\overline{\mu}$-calculus

A Simulations of Milner Abstract Machines with sequent calculi

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

It is quite obvious that the call-by-name $\overline{\lambda}\mu\overline{\mu}$-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 $\overline{\lambda}\mu\overline{\mu}$-calculus:

$$
\llbracket t \parallel \pi \parallel \tau \rrbracket \triangleq \llbracket \llbracket t \rrbracket_1 \parallel \llbracket \pi \rrbracket \parallel \llbracket \tau \rrbracket \rrbracket
$$

with:

$$
\begin{align*}
\llbracket x \rrbracket_1 & \triangleq x \\
\llbracket \lambda x . t \rrbracket_1 & \triangleq \lambda x . \llbracket t \rrbracket_1 \\
\llbracket \mu a . \overline{\lambda} x . \llbracket t \rrbracket_1 \parallel \llbracket \pi \rrbracket & \triangleq \mu a . \langle \llbracket t \rrbracket_1 \parallel \llbracket \pi \rrbracket \parallel \llbracket \alpha \rrbracket \rangle
\end{align*}
$$

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

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

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

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). \textbf{If} $\mathcal{S}$, $\mathcal{S}'$ \textbf{are states of the MAM such that} $\mathcal{S} \overset{1}{\rightarrow} \mathcal{S}'$, \textbf{then there exists a closure} $c' \tau'$ \textbf{such that we have} $\llbracket \mathcal{S} \rrbracket \overset{1}{\rightarrow} c' \tau'$ \textbf{and} $(c' \tau') \equiv \llbracket \mathcal{S}' \rrbracket$.

\textbf{Proof.} Trivial induction on reduction rules of the MAM. \hfill $\square$

In other words, we showed that a variant of the $\overline{\lambda}\mu\overline{\mu}$-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 $\overline{\lambda}|t;\tau|\star$-calculus

Similarly, we can prove that the $\overline{\lambda}|t;\tau|\star$-calculus allows us to simulate reductions of the MAD. The compilation function from states of the MAD to closures of the $\overline{\lambda}|t;\tau|\star$-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 $\overline{\lambda}|t;\tau|\star$-calculus through the binder $\overline{\mu}[x].\langle x \parallel F \rangle r'$.

Following this intuition, the definition of the translation is almost direct:

$$
\llbracket t \parallel \pi \parallel \tau \parallel \overline{D} \rrbracket \triangleq \llbracket \llbracket t \rrbracket_1 \parallel \llbracket \pi \rrbracket \parallel \llbracket \tau \rrbracket \parallel \llbracket \overline{D} \rrbracket \rrbracket
$$
with:

\[
\begin{align*}
[x], & \triangleq x \\
[\lambda x.t], & \triangleq \lambda x.[t], \\
[tv], & \triangleq \mu \alpha. ([t], [u]_t \cdot \alpha)
\end{align*}
\]

\[
\begin{align*}
[u \cdot \pi]_\pi & \triangleq [u]_t, \\
[u]_\pi & \triangleq [u]_t \\
[t[x := t]]_\pi & \triangleq [t]_\pi [x := [t]_t] \\
[t]_\pi & \triangleq \epsilon
\end{align*}
\]

\[
\langle x | | J \pi K \rangle \triangleq [J \pi K]_\pi \\
\begin{align*}
\epsilon & \triangleq \kappa
\end{align*}
\]

Once again, it is straightforward to check that:

**Proposition A.2 (MAD simulation).** *If S, S’ are two states of the MAD such that S \rightarrow S’, then there exists a closure c’\tau’ such that we have [S] \rightarrow c’\tau’ and (c’\tau’) \equiv [S’].*

*Proof.* Trivial induction on reduction rules of the MAD. \(\square\)
A calculus of expandable stores

Hugo Herbelin and Étienne Miquey

B.1 Definitions

| Strong values | $\nu ::= \lambda x.t \mid k$ |
| Weak values   | $V ::= \nu \mid x$ |
| Terms         | $t, u ::= V \mid \mu x.c$ |
| Environments  | $\tau ::= \epsilon \mid \tau[x := t] \mid \tau[\alpha := E]$ |
| B. Definitions | |
| Forcing contexts | $F ::= t \cdot E \mid k$ |
| Catchable contexts | $E ::= F \mid \alpha \mid \hat{\mu}[x].(x \parallel F)\tau$ |
| Evaluation contexts | $e ::= E \mid \hat{\mu}x.c$ |

(BLET) \[
\Gamma \vdash \lambda x.t \mid \hat{\mu}x.e \tau \rightarrow \Gamma \vdash [x := t] e C\tau
\]

(CATCH) \[
\Gamma \vdash \mu x.e \parallel E \tau \rightarrow \Gamma \vdash [x := E] x \parallel E \tau
\]

(LOOKUP,) \[
\Gamma \vdash \nu \parallel E \tau \rightarrow \Gamma \vdash [x := \nu] \parallel E \tau
\]

(LOOKUP, ) \[
\Gamma \vdash \nu \parallel F \tau[x := t] \tau \rightarrow \Gamma \vdash \nu \parallel [x \parallel F] \tau[x := t] \tau
\]

(RESTORE) \[
\Gamma \vdash \nu \parallel \hat{\mu}[x].(x \parallel F) \tau \rightarrow \Gamma \vdash \nu \parallel [x \parallel F] \tau
\]

(BETA) \[
\Gamma \vdash \lambda x.t \parallel u \cdot E \tau \rightarrow \Gamma \vdash [u \parallel \hat{\mu}x.(t \parallel E)] \tau
\]

Figure 12. The $\lambda[\nu\tau*]$-calculus

B. The $\lambda[\nu\tau*]$-calculus

We recall here the definition of Ariola et al.’s $\lambda[\nu\tau*]$-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 $\alpha$-renaming

The original presentation of the $\lambda[\nu\tau*]$-calculus deeply relies on the assumption that names of variable are unique and thus on the possibility of performing $\alpha$-conversion on-the-fly. Consider for instance a command formed by a term of the shape $t = \mu x.\langle u \parallel \hat{\mu}x.(x \parallel \alpha) \rangle$ and a context of the shape $e = \hat{\mu}x.(x \parallel F)$. Such a command is perfectly typable (if $u$ and $F$ are) in the type system introduced in [26], however, reducing this command (without $\alpha$-conversion) would loop forever because of the
auto-reference \([x := x]\) in the environment:

\[
\begin{align*}
\langle \mu \alpha. \langle u \parallel \mu x. (x \parallel \alpha) \rangle \rangle &\mapsto (x \parallel F)[x := \mu \alpha. \langle u \parallel \mu x. (x \parallel \alpha) \rangle] \\
&\rightarrow \langle \mu \alpha. \langle u \parallel \mu x. (x \parallel \alpha) \rangle \rangle \parallel \mu [x]. (x \parallel F) \\
&\rightarrow \langle u \parallel \mu x. (x \parallel \alpha) \rangle[\alpha := \mu [x]. (x \parallel F)] \\
&\rightarrow (x \parallel \alpha)[\alpha := \mu [x]. (x \parallel F), x := u] \\
&\rightarrow (x \parallel \mu [x]. (x \parallel F))[\alpha := \mu [x]. (x \parallel F), x := u] \\
&\rightarrow (x \parallel F)[\alpha := \mu [x]. (x \parallel F), x := u, x := x] \rightarrow \ldots
\end{align*}
\]

While a simple \(\alpha\)-conversion of one of the \(x\) binding solves the problem, this becomes much more subtle to handle through a CPS translation without renaming (as the one in Figure 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 \(\alpha\)-conversion on the fly during the translation. Thus, the problem becomes unsolvable after the translation, and the problem of renaming should be tackled together with the definition of the translation of terms. For
instance, through this translation, the same closure is again a program that will loop forever:

\[
\begin{align*}
\llbracket e \rrbracket = & \llbracket e \rrbracket_\alpha \in \llbracket I \rrbracket \Rightarrow \llbracket \bar{\mu}x.(x \parallel F) \rrbracket_\alpha \in \llbracket I \rrbracket \\
= & \llbracket (x \parallel F) \rrbracket_\alpha \cdot \llbracket x := \llbracket I \rrbracket_\alpha \rrbracket \\
= & \llbracket x \rrbracket_\alpha \cdot \llbracket x := \llbracket I \rrbracket_\alpha \rrbracket \parallel \llbracket F \rrbracket_\alpha \\
= & \llbracket \mu \alpha.t \parallel \bar{\mu}x.(x \parallel \alpha) \rrbracket_\alpha \parallel \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha \\
= & \llbracket (\mu \alpha.t \parallel \bar{\mu}x.(x \parallel \alpha)) \rrbracket_\alpha \parallel \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha, \llbracket x := \llbracket u \rrbracket_\alpha \rrbracket \\
= & \llbracket \mu \alpha.t \parallel \bar{\mu}x.(x \parallel \alpha) \rrbracket_\alpha \parallel \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha, \llbracket x := \llbracket u \rrbracket_\alpha \rrbracket \parallel \llbracket x \rrbracket_\alpha \\
= & \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha, \llbracket x := \llbracket u \rrbracket_\alpha \rrbracket \parallel \llbracket x \rrbracket_\alpha \\
\rightarrow & \llbracket x \rrbracket_\alpha \parallel \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha, \llbracket x := \llbracket u \rrbracket_\alpha \rrbracket \parallel \llbracket x \rrbracket_\alpha
\end{align*}
\]

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

In order to ensure the correctness of our translation, we address the problem at the source in the \(\overline{\lambda_{\mu\tau\tau\alpha\alpha}}\) using de Bruijn levels. As we observed in the previous example, the issue arises when adding a binding \([x := \ldots]_\alpha\) 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 at which they occur in the environment, which is obviously unique. Before presenting formally the corresponding system and the adapted translation, let us reduce the same example using this idea. We use a mixed notation for names, writing \(x\) when a variable is bound by a \(\lambda\) or a \(\bar{\mu}\), 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:

\[
\begin{align*}
\llbracket \mu \alpha.t \parallel \bar{\mu}x.(x \parallel \alpha) \rrbracket_\alpha \parallel \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha & \rightarrow \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha, \llbracket x := \llbracket u \rrbracket_\alpha \rrbracket \parallel \llbracket x \rrbracket_\alpha \\
\llbracket \mu \alpha.t \parallel \bar{\mu}x.(x \parallel \alpha) \rrbracket_\alpha \parallel \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha & \rightarrow \llbracket \lambda \tau \lambda V.V \cdot \tau \parallel \lambda \tau \lambda E.E \cdot \tau \parallel \llbracket F \rrbracket_\alpha, \llbracket x := \llbracket u \rrbracket_\alpha \rrbracket \parallel \llbracket x \rrbracket_\alpha \\
\end{align*}
\]

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 \(\llbracket - \rrbracket_\sigma\) where \(\sigma\) is a substitution used to rename variables. To compact the notations, we write \(\llbracket x_i^0 \rrbracket_{\sigma} \ldots \llbracket x_i^m \rrbracket_{\sigma}\) for the renaming substitution \([x := m, \alpha := y, \ldots, \gamma]_\sigma\), where we adopt the convention that the most recent binding is on written on the right. As a binding \([x := n]_\sigma\) overwrites any former binding \([x := m]_\sigma\), we write \([x_i^0]_{\overline{\sigma}}\) instead of \([x_i^0]_{\sigma^m_{\overline{x_i}}}\). Using this trick, the translation of the former command would be (where \(m\) and \(n\)})
are fresh names generated during the translation:

\[ \begin{align*}
\llbracket ce \rrbracket^e_t &= \llbracket e \rrbracket^e_t \cdot \llbracket e \rrbracket^c_t \\
\llbracket xu. (x \parallel F) \rrbracket^e_t &= [\llbracket xu. (x \parallel F) \rrbracket^e_t]_{\llbracket e \rrbracket^c_t} \\
\llbracket xu. (x \parallel F) \rrbracket^e_t &= [\llbracket xu. (x \parallel F) \rrbracket^e_t]_{\llbracket e \rrbracket^c_t}
\end{align*} \]

By definition of the lifted term \( \llbracket tu \rrbracket^e_t \), the term \( t \) where all the free variables \( x_j \) with \( j \geq i \) (resp. \( a_j \)) have been replaced by \( x_{j+n} \) (resp. \( a_{j+n} \)). Given \( i, n \) two natural numbers, we define:

\[ \begin{align*}
\uparrow_i^n (ct) &= \Delta (\uparrow_i^n c)(\uparrow_i^n t) \\
\uparrow_i^n (t \parallel e) &= \Delta (\uparrow_i^n t \parallel \uparrow_i^n e) \\
\uparrow_i^n (\kappa) &= \Delta \kappa \\
\uparrow_i^n (t \cdot E) &= \Delta (\uparrow_i^n t) \cdot (\uparrow_i^n E) \\
\uparrow_i^n (x_j) &= \Delta a_j \quad \text{(if } j < i \text{)} \\
\uparrow_i^n (a_j) &= \Delta a_{j+n} \quad \text{(if } j \geq i \text{)} \\
\uparrow_i^n (\mu x_j). (x_j \parallel F) \tau &= \Delta \mu[\uparrow_i^n x_j]. (\uparrow_i^n (x_j \parallel F) \tau) \\
\uparrow_i^n (\mu x_j). c &= \Delta \mu[\uparrow_i^n x_j]. (\uparrow_i^n c)
\end{align*} \]

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 \( \uparrow_i^n t \), the term \( t \) where all the free variables \( x_j \) with \( j \geq i \) (resp. \( a_j \)) have been replaced by \( x_{j+n} \) (resp. \( a_{j+n} \)). Given \( i, n \) two natural numbers, we define:

\[ \begin{align*}
\uparrow_i^n (ct) &= \Delta (\uparrow_i^n c)(\uparrow_i^n t) \\
\uparrow_i^n (t \parallel e) &= \Delta (\uparrow_i^n t \parallel \uparrow_i^n e) \\
\uparrow_i^n (\kappa) &= \Delta \kappa \\
\uparrow_i^n (t \cdot E) &= \Delta (\uparrow_i^n t) \cdot (\uparrow_i^n E) \\
\uparrow_i^n (x_j) &= \Delta a_j \quad \text{(if } j < i \text{)} \\
\uparrow_i^n (a_j) &= \Delta a_{j+n} \quad \text{(if } j \geq i \text{)} \\
\uparrow_i^n (\mu x_j). (x_j \parallel F) \tau &= \Delta \mu[\uparrow_i^n x_j]. (\uparrow_i^n (x_j \parallel F) \tau) \\
\uparrow_i^n (\mu x_j). c &= \Delta \mu[\uparrow_i^n x_j]. (\uparrow_i^n c)
\end{align*} \]
C Context-free abstract machines

C.1 The named context-free abstract machine for the $\lambda_{\text{ref}}$-calculus

\[
\begin{align*}
\langle t \parallel x \cdot c \rangle_{e\tau} &\rightarrow c_{e\tau}[x := t] \\
\langle t \parallel E \rangle_{e\tau} &\rightarrow \langle t \parallel E \rangle_{e\tau} \\
\langle \mu x.c \parallel E \rangle_{t\tau} &\rightarrow c_{t\tau}[\alpha := E] \\
\langle V \parallel E \rangle_{t\tau} &\rightarrow \langle V \parallel E \rangle_{t\tau} \\
\langle V \parallel \mu x.E \rangle_{t\tau} &\rightarrow \langle V \parallel E \rangle_{t\tau} \\
\langle x \parallel F \rangle_{v\tau} &\rightarrow \langle x \parallel F \rangle_{v\tau} \\
\langle v \parallel u \cdot E \rangle_{v\tau} &\rightarrow \langle v \parallel u \cdot E \rangle_{v\tau} \\
\langle \lambda x.x \parallel u \cdot E \rangle_{v\tau} &\rightarrow \langle \mu x.(x \parallel u \cdot E) \parallel E \rangle_{v\tau}
\end{align*}
\]

Figure 15. Context-free abstract machine for the $\lambda_{\text{ref}}$-calculus [5]

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

\[
\begin{align*}
\langle t \parallel x_i.c \rangle_{e\tau} &\rightarrow c_{e\tau}[x_i := t] \quad \text{with } |\tau| = n \\
\langle t \parallel E \rangle_{e\tau} &\rightarrow \langle t \parallel E \rangle_{e\tau} \\
\langle \mu x_i.c \parallel E \rangle_{t\tau} &\rightarrow c_{t\tau}[\alpha_i := E] \quad \text{with } |\tau| = n \\
\langle x_i \parallel E \rangle_{t\tau} &\rightarrow \langle t \parallel E \rangle_{t\tau} \\
\langle V \parallel E \rangle_{t\tau} &\rightarrow \langle V \parallel E \rangle_{t\tau} \\
\langle V \parallel \alpha_i.E \rangle_{t\tau} &\rightarrow \langle V \parallel \alpha_i.E \rangle_{t\tau} \\
\langle V \parallel u \cdot E \rangle_{v\tau} &\rightarrow \langle V \parallel u \cdot E \rangle_{v\tau} \\
\langle \lambda x.t \parallel u \cdot E \rangle_{v\tau} &\rightarrow \langle \mu x.(x \parallel u \cdot E) \parallel E \rangle_{v\tau}
\end{align*}
\]

Figure 16. Context-free abstract machine for the call-by-name $\lambda\mu\tilde{\mu}$-calculus with environments
We can in fact safely reduce properties of computable coercions to the ones of normal forms:

\[ \Gamma \vdash \sigma_0 : Y_0' \prec Y_0 \quad \Gamma \vdash \sigma_1 : Y'_1 \prec Y_1 \]
\[ \Gamma \vdash \sigma : Y' \prec Y \]

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:

\[ \sigma ::= \varepsilon \mid \sigma^+ \mid \|\sigma \mid \sigma' \circ \sigma \]

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 \( \rightarrow_{\sigma} \). In other word, we can define the composition function \( \circ \circ \) between coercions in normal form by:

\[ \sigma^+_1 \circ \sigma^+_0 \triangleq (\sigma_1 \circ \sigma_0)^+ \]
\[ \uparrow \sigma_1 \circ \sigma_0 \triangleq \| (\sigma_1 \circ \sigma_0) \]
\[ \sigma^+_1 \circ \| \sigma_0 \triangleq \| (\sigma_1 \circ \sigma_0) \]
\[ \varepsilon \circ \sigma_0 = \sigma_1^+ \circ \varepsilon \triangleq \varepsilon \]

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

\[ \quad \]

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

\[ \sigma ::= \varepsilon \mid \sigma^+ \mid \|\sigma \mid \sigma' \circ \sigma \]

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

\[ \Gamma \vdash \sigma : Y' \prec Y \]
\[ \text{then } \vdash \sigma' \circ \sigma : Y' \prec Y' \]

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\(^\text{17}\). We then define computable coercions as being coercions of the shape:

\[ \sigma ::= \varepsilon \mid \sigma^+ \mid \|\sigma \mid \sigma' \circ \sigma \]

\[ \text{Proposition D.3 (Computing normal forms). For any computable } \sigma, \text{ if } \Gamma \vdash \sigma : Y < Y' \text{ then there exists } \sigma_n \text{ in normal form such that } \Gamma \vdash \sigma_n : Y' < Y. \]

\[ \text{Proof. By induction on typing derivations, using the previous lemma for the } (\prec_{\sigma}) \text{-rule.} \]

It is worth noting that for any \( \sigma, Y, Y' \), if \( \vdash \sigma : Y' < Y \), then \( \sigma \) is necessarily computable.

\[ \text{Corollary D.4. If } \vdash \sigma : Y' < Y, \text{ then } |Y| \leq |Y'|. \]

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

\[ \text{17We could also include } \|_{\tau} \sigma_n \text{ and } \sigma_{n+1} \text{ where } \tau \text{ is in normal form, but for the sake of simplicity, let us just focus on this simpler fragment.} \]
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 $\sigma$ for the coercion $\uparrow((\uparrow t)^{++})$ which is in normal form, we have:

\begin{itemize}
  \item $\vdash \uparrow((\uparrow t)^{++}) : T_0, T, U, T_1 <: T, U$
  \item $\text{dom}(\sigma) = 2, \text{codom}(\sigma) = 4$
  \item $\|\sigma\| = \begin{cases} 
    0 &\mapsto 1 \\
    1 &\mapsto 2 \\
    2 &\mapsto 4 
  \end{cases}$
\end{itemize}

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 $\sigma$ is in normal form and $\vdash \sigma : Y' <: Y$, then:

1. $\text{dom}(\sigma) = |Y|$
2. $\text{codom}(\sigma) = |Y'|$
3. $\forall n < |Y|, Y'(\|\sigma\|(n)) = Y(n)$
4. $\|\sigma\|(\|Y\|) = |Y'|$

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 $(<_{\text{ax}})$ is trivial; while for $(<_{\text{as}})$ and $(<_{\text{gs}})$ it suffices to unfold the definition, using the first items to connect $|Y|$ and $\text{dom}(\sigma)$.

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

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

1. For any $Y$, we have $\vdash \text{id}_Y : Y <: Y$ (\text{<}_{\text{id}})
2. If $\vdash \sigma : Y <: Y'$ and $\vdash \sigma' : Y' <: Y''$, then $\vdash \sigma' \circ \sigma : Y <: Y''$.
3. If $\vdash \sigma : Y <: Y'$ and $\vdash \sigma' : Y' <: Y$, then $Y = Y'$.

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 $[0, |Y|]$ to itself (by Corollary D.4 we have $|Y'| = |Y|$), they are necessarily the identity. Equivalently, we could have seen that necessarily $\sigma$ is of the shape $\sigma_0$ (and so is $\sigma'$), so that $Y' = Y_0$, $T$ and $Y = Y_0, T$, from which we can conclude by an easy induction.

\section{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:

\[
\begin{array}{c}
\Gamma \vdash t : A \\
\Gamma' \vdash \Gamma \subseteq \Gamma' \quad (\text{w})
\end{array}
\]

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

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 $\sigma$, by precomposing terms with $\sigma$. Remember that given a coercion $\sigma$ of type $Y' <: Y$ and a term $t$ whose type is of the shape $\forall Y <: Y.A$, we defined:

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

Lemma D.9. The following rule is admissible:

\[
\begin{array}{c}
\Gamma \vdash t : \forall Y <: Y.A \\
\Gamma \vdash \uparrow t^\sigma : \forall Y <: \uparrow Y'.A \quad (\text{w})
\end{array}
\]

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

\[
\begin{array}{c}
\Gamma \vdash t : \forall Y <: Y_0.A \\
\Gamma \vdash t.Y : Y <: Y_0.A \quad (\text{w}) \\
\Gamma, s : Y <: Y_1 \vdash t.Y : Y <: Y_0.A \quad (\text{w}) \\
\Gamma, s : Y <: Y_1 \vdash t.Y : Y_1 \vdash s : X <: Y_1 \quad (\text{w})
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \lambda Y.s.t Y (s \circ \sigma) : A \\
\Gamma \vdash \lambda Y.s.t Y (s \circ \sigma) : \forall Y <: Y_1.A \quad (\text{w})
\end{array}
\]

\section{D.3 Subject expansion}

This is a dual result of subject reduction. Intuitively, a subject expansion rule should say that a term $t$ of type $\forall Y : T.A$ may be replaced by a term $t^\sigma$ of type $\forall Y : \uparrow Y'.A$ as long as $\uparrow Y'$ is a subtype of $Y$. Formally, the rule can be expressed as:

\[
\begin{array}{c}
\Gamma \vdash t : \forall Y : T.A \\
\Gamma \vdash \uparrow t^\sigma : \forall Y : \uparrow Y'.A \quad (\text{w})
\end{array}
\]

The key step in the proof is to define the coercion $\sigma$ in such a way that $\uparrow Y'$ is a subtype of $Y$. This is achieved by using the fact that $\sigma$ is a function from $\Gamma$ to $\Gamma'$, $\lambda X.s.t X (s \circ \sigma)$, so that $\uparrow Y'$ is a subtype of $Y$. The proof then follows by induction on typing derivations, using the subject reduction rule to reduce $\uparrow t^\sigma$ to $t$. The details of the proof are left as an exercise for the reader.
where we use Lemma D.8 to weaken \( \Gamma, \sigma : X <: Y_1 \).

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

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

\[
\Gamma \vdash t : Y_0 \triangleright F \quad \Gamma \vdash \sigma : Y_1 <: Y_0 \\
\Gamma \vdash \tau : \lambda t'. Y_1 \triangleright F
\]

**Lemma D.11.** If \( \sigma_1 \) is in normal form, then the following rule is admissible:

\[
\Gamma \vdash \sigma_0 : Y_0' <: Y_0 \quad \Gamma \vdash \sigma_1 : Y_1' <: Y_1 \\
\Gamma \vdash \sigma_0; \sigma_1 : Y_0'; Y_1' <: Y_0; Y_1
\]

We can now verify type safety with respect to reductions:

**Theorem 3.4** (Subject reduction). For any context \( \Gamma \), any type \( T \) and any terms \( t, t' \), if \( \Gamma \vdash t : T \) and \( t \rightarrow t' \), 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 \( \Gamma \vdash \tau : \Delta \), then \( \Gamma, \Gamma' \vdash \tau[u/x] : B \).
2. If \( \Gamma, \sigma : \psi \vdash \tau, \Gamma' \vdash t : B \) and \( \Gamma \vdash \tau : \gamma; \gamma', \Gamma' \vdash t[\tau/\delta] : B \).
3. If \( \Gamma, \tau : \psi; \Gamma \vdash t : B \) and \( \Gamma \vdash \sigma : \psi' ; \Gamma' \vdash t[\sigma/s] : B[\psi'/\psi] \).
4. If \( \Gamma \vdash t : B \) then \( \Gamma[\psi'/\psi] \vdash t[\psi'/\psi] : B[\psi'/\psi] \).

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:

\[
\Gamma \vdash \sigma : Y_1 <: Y_0 \\
\Gamma \vdash \tau : \lambda t'. Y_1 \triangleright F \\
\Gamma \vdash \tau \triangleright t \triangleright t' \rightarrow \Gamma \vdash \tau' \triangleright t' \triangleright t
\]

Indeed, if we have:

\[
\Gamma \vdash \sigma : Y_1 <: Y_0 \\
\Gamma \vdash \tau : \lambda t'. Y_1 \triangleright F \\
\Gamma \vdash \tau \triangleright t \triangleright t' \rightarrow \Gamma \vdash \tau' \triangleright t' \triangleright t
\]

we can derive (using Corollary D.10):

\[
\Gamma \vdash \sigma : Y_1 <: Y_0 \\
\Gamma \vdash \tau : \lambda t'. Y_1 \triangleright F \\
\Gamma \vdash \tau \triangleright t \triangleright t' \rightarrow \Gamma \vdash \tau' \triangleright t' \triangleright t
\]

\[
\Gamma \vdash \tau \triangleright t \triangleright t' \rightarrow \Gamma \vdash \tau' \triangleright t' \triangleright t
\]

\[
\Gamma \vdash \tau \triangleright t \triangleright t' \rightarrow \Gamma \vdash \tau' \triangleright t' \triangleright t
\]

\[
\Gamma \vdash \tau \triangleright t \triangleright t' \rightarrow \Gamma \vdash \tau' \triangleright t' \triangleright t
\]

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

\[
\Gamma \vdash \tau : Y' \quad \Gamma \vdash \sigma : Y' <: Y \quad Y = [\Gamma_0]; \mathcal{F}, [\Gamma]; \quad |\Gamma_0| = n \\
\Gamma \vdash \text{split} \ \tau \ \text{at} \ n \ \text{along} \ \sigma : Y' <: Y \ \text{as} \ (Y_0, s_0, \delta_0, x, (Y_1, s_1, \delta_1)) \ \text{in} \ t : A'
\]

where \( \Gamma' = \Gamma[\psi'; \psi] \) and \( A' = \lambda[\psi' \psi'] \) (which is an abuse of notations to say that \( \Gamma' = \Gamma[\psi'/\psi] \), \( \Gamma = \Gamma[\psi' \psi'] \) for some \( \Gamma_0, \) etc). The latter equalities are necessary for identifying \( Y' \) and its cutting in the typing derivation for \( \tau \), which may contain at the same time build on \( s_0, s_1, \ldots \) thus referring to \( Y_0, Y_1 \) and terms referring directly to \( Y' \). 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 \( s_1 : Y_0; F; Y_1 <: Y' \) and \( s_2 : Y' <: Y_0; F; Y_1 \) in the premise. In addition to giving access to subterm in \( t \) to both representations of \( Y' \), it also enforces the equality between \( Y' \) and \( Y_0, F; Y_1 \) (remember that \( <: \) induces an order relation, Proposition D.7). In particular, the reduction rule for \text{split} should then instantiate those two coercions with \( \text{id}_Y \). This approach would result in the following typing rule:

\[
\Gamma \vdash \tau : Y' \\
\Gamma \vdash \sigma : Y' <: Y \quad Y = [\Gamma_0]; \mathcal{F}, [\Gamma]; \quad |\Gamma_0| = n \\
\Gamma \vdash \text{split} \ \tau \ \text{at} \ n \ \text{along} \ \sigma : Y' <: Y \ \text{as} \ (Y_0, s_0, \delta_0, x, (Y_1, s_1, \delta_1), s_i, s_j) \ \text{in} \ t : A'
\]
together with the following reduction rule:

\[
\text{split } r_0[u]; r_1 \text{ at } n \text{ along } \sigma : Y_0, A, Y_1 <: \llbracket I_0 \rrbracket, A, \llbracket I_1 \rrbracket \text{ as } (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1), s_i, s_j \text{ in } t \rightarrow t[Y_0/Y_0](\sigma_0/s_0)[r_0/\delta_0][u/x][Y_1/Y_1](\tau_1/\delta_1)[\sigma_1/s_1][id_{\delta_0,A,Y_1/s_i}][id_{\delta_0,A,Y_1/s_j}]
\]

where \( n = |I_0|, |r_0| = |Y_0| \) and \( Y_0, Y_1, \sigma_0, \sigma_1 \) are again as in Lemma 3.2. This reduction is also safe with respect to typing.

D.3 Normalization

The proof of normalization for \( F_\lambda \) 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:

\[
\begin{align*}
\text{Terms} & \quad t & := & \quad k | x | \lambda Y.t | t Y | \lambda s.t | t \sigma | \lambda \delta . t | t \tau | \lambda x . t | t u \\
\text{Stacks} & \quad \pi & := & \quad \cdot | Y . \pi | \sigma . \pi | \tau . \pi | t . \pi \\
\text{Processes} & \quad p & := & \quad t \star \pi
\end{align*}
\]

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

\[
\begin{align*}
    t M \star \pi & > t \star M . \pi \\
    \lambda Y.t \star Y . \pi & > t[Y/Y] \star \pi \\
    \lambda s.t \star \sigma . \pi & > t[\sigma/Y] \star \pi \\
    \lambda \delta . t \star \tau . \pi & > t[\tau/Y] \star \pi \\
    \lambda x . t \star u . \pi & > t[u/Y] \star \pi
\end{align*}
\]

to which we add a rule for splitting stores:

\[
\text{split } r_0[u]r_1 \text{ at } n \text{ along } \sigma : Y_0, A, Y_1 <: \llbracket I_0 \rrbracket, A, \llbracket I_1 \rrbracket \text{ as } (Y_0, s_0, \delta_0), x, (Y_1, s_1, \delta_1), s_i, s_j \text{ in } t \star \pi \rightarrow t[Y_0/Y_0](\sigma_0/s_0)[r_0/\delta_0][u/x][Y_1/Y_1](\tau_1/\delta_1)[\sigma_1/s_1] \star \pi
\]

where \( n = |I_0| \) and \( Y_0, Y_1, \sigma_0, \sigma_1 \) are as in Lemma 3.2 (i.e. \( |I_0| = |\sigma|(n), \sigma_0 : Y_0 < I_0, \) etc.), and \( |r_0| = |Y_0| \).

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

\[
\begin{align*}
    \text{split } r \text{ at } n \text{ along ... as ... in } t \star \pi & > \text{split } r' \text{ at } n \text{ along ... as ... in } t \star \pi & \quad (\text{if } \tau \rightarrow \tau') \\
    \text{split } r_0 \text{ at } n \text{ along } \sigma : ... \text{ as ... in } t \star \pi & > \text{split } r_0' \text{ at } n \text{ along } \sigma' : ... \text{ as ... in } t \star \pi & \quad (\text{if } \sigma \rightarrow \sigma')
\end{align*}
\]

where \( r_0 \) 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 \( \bot \subseteq \Lambda \star \Pi \) is said to be closed under anti-reduction whenever for all closed \( p, p' \in \Lambda \star \Pi \), if \( p' \in \bot \) and \( p > p' \) then \( p \in \bot \). A pole is defined as any set of closed processes that is closed under anti-reduction.

Definition D.13 (Valuation). A substitution, written \( \rho \), 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 \( \rho, Y := Y \)).

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

\[\text{See for instance the proof of normalization for system } D \text{ presented in [17, 3.2].}\]
We say that a typing rule \( \tau \) is adequate if the validity of the premises \( \rho \vdash \tau \) entails the validity of the conclusion \( \rho \vdash \tau' \).

Given a closed context \( \Gamma \), we say that a substitution \( \rho \) realizes \( \Gamma \), which we write \( \rho \vdash \Gamma \), if:

1. for any \( Y \in \Gamma \), \( \rho(Y) \) is defined
2. for any \( (s : Y' <: Y) \in \Gamma \), we have \( \rho(s) \vdash \rho(Y') \vdash \rho(Y) \)
3. for any \( (\delta : Y' \triangleright \tau, Y) \in \Gamma \), we have \( \rho(\delta) \vdash \rho(Y') \triangleright \rho(\tau) \)
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. \( \Gamma \vdash \sigma : Y' <: Y \), \( \Gamma \vdash \tau : Y' \triangleright \tau, Y \)) is valid with respect to the interpretation if for any substitution \( \rho \) such that \( \rho \vdash \Gamma \), then \( \rho(t) \vdash \rho(A) \) (resp. \( \rho(s) \vdash \rho(Y') \vdash \rho(Y) \), \( \rho(\delta) \vdash \rho(Y') \triangleright \rho(\tau) \)).

We say that a typing rule \( \frac{J_1 \ldots J_n}{\J} \) is adequate if the validity of the premises \( J_1, \ldots, J_n \) 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 readability, we write \( t_\rho \) (resp. \( A_\rho \)) for \( \rho(t) \) (resp. \( \rho(A) \)).

\[(x : A) \in \Gamma \]

- Case \( \frac{\Gamma \vdash t : A}{\Gamma \vdash x : A}^{(Ax)} \). By definition, since \( \rho \vdash \Gamma \) and \( (x : A) \in \Gamma \), we have that \( \rho(x) \vdash \rho(A) \).

\[\Gamma, x : A \vdash t : B \]

- Case \( \frac{\Gamma \vdash \lambda x.t : A \Rightarrow B}{\Gamma \vdash \lambda x.t : A}^{(\lambda)} \). Let \( u \cdot \pi \in ||A \Rightarrow B|| \). \( \Gamma \vdash \lambda x.u \cdot \pi \Rightarrow \tau, \pi \in ||A \Rightarrow B|| \). Then \( \lambda x.t_\rho \star u \cdot \pi \tau \rho[u/x] \cdot \pi \in \|A \Rightarrow B\| \) 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.

\[\frac{\Gamma \vdash t : A \Rightarrow B, u : B}{\Gamma \vdash t u : B}^{(\circ)} \]

- Case \( \frac{\Gamma \vdash t : A \Rightarrow B, u : B}{\Gamma \vdash t u : B} \). Let \( \pi \in ||B|| \), by induction hypothesis we have \( t_\rho \vdash A \Rightarrow B \) and \( u_\rho \vdash A \). Therefore, \( t_\rho \star u_\rho \pi \star \tau_\rho \pi \rho \star \pi \rho \pi \in \|A \Rightarrow B\| \). We conclude by anti-reduction.

\[\frac{\Gamma \vdash [] : B}{\Gamma \vdash \emptyset \Rightarrow B}^{(\emptyset)} \]

- Case \( \frac{\Gamma \vdash \emptyset}{\Gamma \vdash \emptyset \Rightarrow \emptyset} \). Trivial: for all \( \tau \vdash \emptyset \), we have that \( \tau ; [] = \tau \) and thus \( \tau ; [] \vdash \emptyset \).

\[\frac{\Gamma \vdash t : B_0 \Rightarrow \emptyset, F}{\Gamma \vdash t : B_0 \Rightarrow \emptyset}^{(\top)} \]

- Case \( \frac{\Gamma \vdash \emptyset, F}{\Gamma \vdash t : B_0 \Rightarrow \emptyset} \). Let \( t_{\rho_0} \vdash \emptyset_\rho. \) By induction hypothesis, we have that \( t_\rho \vdash \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow F \). Hence, by definition, \( \top \vdash \emptyset_\rho \Rightarrow F \).

\[\frac{\Gamma \vdash \tau : \emptyset_\rho \Rightarrow \emptyset_\rho, Y \Rightarrow Y'}{\Gamma \vdash \tau : \emptyset_\rho \Rightarrow \emptyset_\rho, Y \Rightarrow Y'}^{(\tau \rho)} \]

- Case \( \frac{\Gamma \vdash \tau_\rho : \emptyset_\rho \Rightarrow \emptyset_\rho, Y \Rightarrow Y'}{\Gamma \vdash \tau \rho : \emptyset_\rho \Rightarrow \emptyset_\rho, Y \Rightarrow Y'} \). Let \( t_{\rho_0} \vdash \emptyset_\rho \). By induction hypothesis, we have that \( t_{\rho_0} \vdash \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow Y \) and thus \( (t_{\rho_0} \vdash \emptyset_\rho) ; t'_\rho \vdash \emptyset_\rho \Rightarrow Y \Rightarrow Y' \). Since \( t_\rho \) is closed, we have that \( t_{\rho_0} ; t'_\rho \vdash \tau (t_{\rho_0} ; t'_\rho) \Rightarrow (t_{\rho_0} ; t'_\rho) \vdash \tau \) (by easy induction the structure of \( t_\rho \)), hence \( t_{\rho_0} ; t'_\rho \vdash \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow Y \Rightarrow Y' \).

- Case \( \vdash (\vdash) \) rules. By definition, in all cases we have that \( \sigma_\rho \) is closed, hence \( \vdash \sigma_\rho : Y' \Rightarrow \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow Y \) is well-typed, thus \( \sigma_\rho \vdash Y' \Rightarrow \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow \emptyset_\rho \Rightarrow Y \).
According to the third item above, this process belongs to the pole, and we can thus conclude by anti-reduction.

Case \( \Gamma \vdash \delta \vdash \gamma \vdash \gamma' \). By definition, since \( \rho \vdash \Gamma \) and \((\delta : \gamma \vdash \gamma') \in \Gamma\), we have that \( \rho(\delta) \vdash \delta : \gamma' \).

\[
\Gamma, \delta : \gamma_0 \vdash \gamma \vdash \gamma' \tag{\tau_a}
\]

Case \( \Gamma \vdash \lambda \delta. t : \gamma_0 \vdash \gamma \rightarrow B \). Same proof as for the (\( \lambda \)) case.

\[
\Gamma \vdash t : \gamma_0 \vdash \gamma \rightarrow B \quad \Gamma \vdash \tau : \gamma_0 \vdash \gamma \tag{\tau_b}
\]

Case \( \Gamma \vdash t : \forall Y. A \vdash \forall Y. A \). Same proof as for the (\( \lambda \)) case.

\[
\Gamma \vdash t : \forall Y. A \vdash \forall Y. A \tag{\forall_b}
\]

Proposition D.16. The set \( \mathbb{P} = \{ p : p \text{ normalizes} \} \) is a valid pole. Besides, the typing rule (c) is adequate for this pole

Proof. Easy verification: if \( p > p' \) and \( p' \) normalizes, then \( p \) normalizes too. Besides, for any \((k : A) \in S\), if \( \pi \in [A] \) then \( k \star \pi \) is blocked hence belongs to \( \mathbb{P} \).

\( \square \)
E  Shallow embedding in Coq

We give here an overview of our take to define a shallow embedding of $F_1$ 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 $\triangleright$ of types in the store as a parameter, emphasizing that the structure of $F_1$ is actually orthogonal to the choice of this translation.

Regarding coercions, we define them as the constructors of the inductive type defining the relation $\ll$. In other words, we only define the coercions in normal form, since through the shallow embedding variables will be Coq variables, while $\sigma \circ \sigma', \ll_1 \sigma$ and $\sigma_{+1}$ are easily definable as Coq functions. Similarly, $\ll_1 \tau$ is also defined through its reduction rules, that is by induction $\bar{Y}$. That way, the reductions rules easily matches Coq reductions: for instance, observe that $\ll_1 \sigma$ 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.

The full Coq development illustrating these ideas (and proving most of the properties we stated on $F_1$) 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:

\[
\begin{align*}
\Gamma &\vdash \tau : Y_0 \vdash Y & \quad \Gamma &\vdash t : Y_0, Y \vdash T \\
\Gamma &\vdash \tau[t] : Y_0 \vdash Y, T \\
\end{align*}
\]

The same holds for \(\Gamma \vdash t : Y_0, Y \vdash E\) and \(\Gamma \vdash \tau[t] : Y_0 \vdash Y, T\).

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

\[
\begin{align*}
\Gamma &\vdash \tau : Y_0 \vdash Y \\
\Gamma &\vdash \tau[t] : Y_0 \vdash Y, T \\
\end{align*}
\]

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:

\[
\begin{align*}
\Gamma &\vdash V : Y \vdash V \\
\Gamma &\vdash \uparrow V : Y \vdash V \\
\end{align*}
\]

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

\[
\begin{align*}
\Pi_E &\vdash V : Y \vdash V \\
\Pi_E &\vdash \uparrow V : Y \vdash V \\
\end{align*}
\]

We are finally equipped to prove the main theorem:

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

1. If \(\Gamma \vdash \nu \vdash T\) then \([\Gamma \vdash \nu \vdash T]\)
2. If \(\Gamma \vdash F : T\) then \([\Gamma \vdash F : T]\)
3. If \(\Gamma \vdash V : T\) then \([\Gamma \vdash V : T]\)
4. If \(\Gamma \vdash E : T\) then \([\Gamma \vdash E : T]\)
5. If \(\Gamma \vdash t : T\) then \([\Gamma \vdash t : T]\)
6. If \(\Gamma \vdash e : T\) then \([\Gamma \vdash e : T]\)
7. If \(\Gamma \vdash c\) then \([\Gamma \vdash c]\)
8. If \(\Gamma \vdash l\) then \([\Gamma \vdash l]\)
9. If \(\Gamma \vdash \tau\) then \([\Gamma \vdash \tau : \Gamma]\)

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 (\(\nu\)) and (\(\lambda\)) rules in one rule.

1. Strong values \([k]_\nu = k\), which has the desired type by hypothesis.
• Case $\lambda x. t$. In the source language, we have:

$$\Gamma, x : T \vdash t : U \quad [\Gamma] = i$$
$$\Gamma, \Gamma' \vdash \lambda x_i, t : T \rightarrow U$$

Hence, we get by induction a proof $\Pi_\Gamma$ of $[[t]] : [\Gamma, x_i : T] \triangleright_1 U$ and we can derive:

$$\begin{align*}
\Pi_\Gamma & \vdash [[t]] : \forall Y' : [\Gamma, x_i : T]. Y' \rightarrow Y' \triangleright_E U \rightarrow \bot \\
\Pi_\delta & \vdash \lambda Ys \delta uE : [\Gamma] \triangleright_E \bot \quad (\lambda)
\end{align*}$$

where:

- $\Pi_\delta$ is a proof of $E : Y \triangleright_E U \rightarrow [\Gamma] \triangleright_E U$ (derivable according to Lemma D.9);
- $\Pi_\delta$ is a proof of $\delta : Y, u : Y \triangleright_1 T; \vdash \delta[u] : Y, T$ (derivable according to Lemma F.1);
- $\sigma_{T, \Gamma'} \triangleq (\circ [\Gamma'] \triangleright_E \bot)$ and $\Pi_\sigma$ is:

$$\begin{align*}
\Pi_\sigma & \vdash \sigma_{T, \Gamma'} : [\Gamma'] : [\Gamma] \\
\Pi_\sigma & \vdash \text{id}_{[\Gamma'] : [\Gamma]} : [\Gamma'] < [\Gamma] \quad (\circ_1)
\end{align*}$$

2. Forcing contexts

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

$$\Gamma \vdash t : T \quad \Gamma \vdash E : U^\bot$$
$$\Gamma \vdash t : (T \rightarrow U)^\bot$$

Therefore, we have by induction hypothesis $t [[t]] : [\Gamma] \triangleright_1 T$ and $[[E]] : [\Gamma] \triangleright_E U$, so that we can derive:

$$\begin{align*}
\delta : Y, u : Y \triangleright_1 T & \vdash Y \triangleright_1 [t] : [\Gamma] \triangleright_1 T \rightarrow Y' \triangleright_1 U \rightarrow \bot \\
\Pi_\sigma & \vdash \text{id}_{[\Gamma] : [\Gamma]} : [\Gamma] < [\Gamma] \\
\Pi_\sigma & \vdash \text{id}_{[\Gamma] : [\Gamma]} : [\Gamma] < [\Gamma] \quad (\circ_1)
\end{align*}$$

where:

- $\Pi_E$ is a proof of $s : Y < [\Gamma] \vdash (\triangleright_E E)_E : Y \triangleright_E U$, derived from the induction hypothesis for $t$ and Lemma D.9;
- $\Pi_\delta$ is a proof of $s : Y < [\Gamma] \vdash (\triangleright_E E)_E : Y \triangleright_1 T$, derived from the induction hypothesis for $E$ and Lemma D.9;
- $\Pi_\delta$ is the axiom rule $\delta : Y \vdash \delta : Y$;
- $\Pi_\sigma$ is the proof of $\circ \text{id}_Y : Y \vdash Y$. 

3. Weak values

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

$$\begin{align*}
\Gamma \vdash v : T \\
\Gamma \vdash v : T
\end{align*}$$
Hence, we have by induction hypothesis that \( \Gamma \vdash [v]: [\Gamma_0]_\tau \rightarrow T \) and we can derive:

\[
\begin{align*}
F : Y_\tau \tau \rightarrow T ; F : Y_\tau \tau \rightarrow Y_\tau Y_\tau \rightarrow Y_\tau & \rightarrow \Pi_\alpha \quad \text{(\( \alpha \))} \\
F \vdash Y_\tau \tau \rightarrow F ; F \vdash Y_\tau Y_\tau \rightarrow Y_\tau & \rightarrow Y_\tau Y_\tau \rightarrow Y_\tau \rightarrow \bot \quad \text{(\( \beta \))} \\
\Pi_\beta \quad \text{(\( \beta \))} \\
\end{align*}
\]

where:
- \( \Pi_\alpha \) is a proof of \( s : Y \vdash [: [\Gamma_0]] : Y_\tau \rightarrow T \), derivable from the induction hypothesis and Lemma D.9.
- \( \Pi_\beta \) is the axiom rule \( \delta : Y \vdash Y \rightarrow T \).
- \( \Pi_\gamma \) is a proof of \( \vdash Id_\tau : Y \rightarrow Y \) (Proposition D.7).

Case \( [[x]]_\gamma \). In the source language, we have:

\[
\Gamma(i) = (x_i : T) \quad \Pi_\delta
\]

so that \( \Gamma \) is of the form \( \Gamma_0, x_i : T, \Gamma_1 \) with \( i = [\Gamma_0] \). By definition, we have:

\[
[[\Gamma_0, x_i : T, \Gamma_1]_\gamma \vdash \lambda Y \delta F. \text{split at } i \text{ along } (s : Y \vdash [: [\Gamma_0]] : T, [\Gamma_1]) \text{ as } (Y_\sigma, s_0, \delta_0), x, (Y_1, s_1, \delta_1)
\]

\[
\quad \text{in } \quad x \vdash Y_0 \vdash Id_\sigma \quad (\delta_\sigma) \quad \text{as } \quad \delta \vdash Y_0 \vdash \bot \quad \text{(Ax)}
\]

\[
The \quad \Pi_F \quad \text{is the following derivation:}
\]

\[
\Gamma, V : Y_\sigma \tau \rightarrow T ; \vdash V : Y_\sigma \tau \rightarrow (\Gamma_0, Y_0, Y_1) \rightarrow (Y_0, T, T) \rightarrow \bot \quad \text{(VY)}
\]

\[
\Gamma, \delta_\sigma : Y_\sigma \tau \rightarrow T ; \vdash \delta_\sigma : Y_\sigma \tau \rightarrow \bot \quad \text{(\( \alpha \))}
\]

\[
\Gamma, \Pi_F \quad \text{is the following proof, obtained by Lemma D.9:}
\]

\[
\Gamma : (Y_0, T, Y_1) \vdash F_\tau T \quad \Gamma : (Y_0, T, Y_1) \vdash F_\tau T \quad \text{(Ax)}
\]

\[
F_\tau \vdash Y_\sigma \tau \rightarrow Y_\sigma \tau \rightarrow (\sigma_\tau) \quad \text{as } \quad s_\sigma \vdash Y_\sigma \tau \rightarrow Y_\sigma \tau \rightarrow \bot \quad \text{(Ax)}
\]

\[
\Gamma : (Y_0, T, Y_1) \vdash F_\tau T \quad \Gamma : (Y_0, T, Y_1) \vdash F_\tau T \quad \text{(VY)}
\]
• $\Pi_\tau$ is the following derivation, where we use Lemmas F.1 and F.2:

\[
\begin{align*}
\delta_1: & (Y_0, T) \vdash Y_1, s_0 < Y_0, \delta_0: Y_0' \vdash V: Y_0' \vdash T \\
\Pi_\tau: & Y_0' \vdash Y_0 < Y_0 + s_0 \vdash Y_0', T < Y_0, T
\end{align*}
\]

\[
\begin{align*}
\delta_1: & (Y_0, T) \vdash Y_1, s_0 < Y_0, \delta_0: Y_0' \vdash V: Y_0' \vdash T + \delta_0[V] : Y_0' \vdash T \\
\Pi_\tau: & Y_0' \vdash Y_0 < Y_0 + s_0 \vdash Y_0', T < Y_0, T
\end{align*}
\]

4. Catchable contexts

• Case $[F]_E$. This case is similar to the case $[\nu]_v$.

• Case $[\mu x_1. (x_1 \parallel F) \tau']_E$. In the source language, we have:

\[
\Gamma, x_1: T, \Gamma': F : T \parallel \Gamma, x_1: T, \tau' : \Gamma' \mid i
\]

We have by induction hypothesis a proof of $\vdash [\tau']_E : [\Gamma, x_1: T] _\Gamma' \vdash T$ and a proof $\Pi_F$ of $\vdash [F]_E : [\Gamma, x_1: T] _\Gamma' \vdash T$.

We can thus derive:

\[
\begin{align*}
\Pi_\tau: & \Gamma, \Gamma' \vdash \mu x_1. (x_1 \parallel F) \tau' : T \\
\Pi_\tau: & \Gamma, \Gamma' \vdash \mu x_1. (x_1 \parallel F) \tau' : T
\end{align*}
\]

where:

• $\sigma_\tau = (\sigma \circ [\tau]\text{id}_{[\Gamma']})'$, $\sigma_\tau = \sigma_{\tau'[\Gamma']}$ and $\sigma_{\tau'} = \emptyset_{A_{\tau'[\Gamma']}}$

• $\Pi_F$ is the following proof, obtained by Lemma D.9:

\[
\begin{align*}
\Pi_F: & s : Y < [\Gamma, \Gamma'] \vdash s : Y < [\Gamma, \Gamma'] \vdash s : Y < [\Gamma, \Gamma'] \\
\Pi_F: & \vdash F : ([\Gamma], T; [\Gamma']) \vdash T
\end{align*}
\]

5. Terms

• Case $[V]_I$. This case is similar to the case $[\tau]_v$.

• Case $[\mu x_1.c]$. In the $\lambda_{\nu \tau *}$-calculus, we have:

\[
\Gamma, \alpha_1 : T \parallel \tau \vdash \mu x_1.c : \Gamma \mid i
\]

\[
\Gamma, \Gamma' \tau_1, \mu x_1.c : T
\]
By induction, we obtain two proofs of \( \vdash s : [\Gamma, x_i : T^a] \Rightarrow_c \bot \) and we can derive:

\[
\Pi_y \quad \vdash [c]_c : [\Gamma, x_i : T^a] \Rightarrow_c \bot \\
\frac{\vdash s : Y <: [\Gamma, \Gamma'] \vdash (s \oplus \Gamma[| \Gamma | \vdash \text{id}_Y])^{\Gamma} : Y, T <: [\Gamma, T]}{s : Y <: [\Gamma, \Gamma'], \delta : Y \vdash [c]_c : (Y, A^s, \sigma_{T, \Gamma'} : (Y, T^a)) \Rightarrow \bot} \quad (\langle \_, \rangle_e) \\
\frac{\delta : Y \vdash \delta : Y}{\delta : Y, E : Y \Rightarrow_c T \vdash \delta[E] : \Rightarrow_c T} \quad (\Rightarrow_e) \\
\frac{s : Y <: [\Gamma, \Gamma'], \delta : Y, E : Y \Rightarrow_c T \vdash \delta[C]_c : (Y, A^s, \sigma_{T, \Gamma'} \delta[E]_E : \bot)}{s : Y \vdash \forall \lambda Y s \delta E, [C]_c : (Y, A^s, \sigma_{T, \Gamma'} : [\Gamma, \Gamma']) \Rightarrow_T} \quad (\lambda)
\]

where \( \Pi_y \) is as in the first case for \( \lambda x \! t \).

### 6. Contexts

- Case \( \llbracket E \rrbracket_c \). This case is similar to the case \( \llbracket v \rrbracket_V \).
- Case \( \llbracket \mu x_i . c \rrbracket_c \). This case is similar to the case \( \llbracket \mu \alpha_i . c \rrbracket_c \).

### 7. Commands

- Case \( \llbracket (t \mid e) \rrbracket_c \). In the \( \tilde{\lambda}_{[t \mid e \tau]} \)-calculus we have:
  \[
  \frac{\Gamma \vdash t : T \quad \Gamma \vdash e : T^a}{\Gamma \vdash (t \mid e)}
  \]
  thus we get by induction two proofs of \( \vdash [t]_t : [\Gamma] \Rightarrow_T \) and \( \vdash [e]_e : [\Gamma] \Rightarrow e T \). We can then derive:

\[
\frac{\vdash [e]_e : [\Gamma] \Rightarrow e T}{s : Y <: [\Gamma] \vdash [e]_e Y \vdash s : Y <: [\Gamma]} \quad (\langle \_, \rangle_e) \\
\frac{\delta : Y \vdash \delta : Y}{\delta : Y, E : Y \Rightarrow_c T \vdash \delta[E] : \Rightarrow_c T} \quad (\Rightarrow_e) \\
\frac{s : Y <: [\Gamma] \vdash \delta : Y}{s : Y : Y \Rightarrow_c T \vdash \delta[s]_s : [\Gamma]} \quad (\Rightarrow_e) \\
\frac{s : Y <: [\Gamma], \delta : Y \vdash [e]_e Y \vdash s \delta (\forall \lambda y Y s \delta E_s) : [\Gamma] \Rightarrow_c \bot}{s : Y <: [\Gamma]} \quad (\forall)
\]

### 8. Closures

- Case \( \llbracket cr \rrbracket_t \). In the \( \tilde{\lambda}_{[t \mid e \tau]} \)-calculus, we have:
  \[
  \frac{\Gamma \vdash t : T \quad \Gamma \vdash e : T^a}{\Gamma \vdash e t}
  \]
  We thus get by induction two proofs \( \vdash [e t]_t : [\Gamma] \Rightarrow_T \) and \( \vdash [e t]_t : [\Gamma, \Gamma'] \Rightarrow_c \bot \). We can derive:

\[
\frac{\vdash [c]_c : [\Gamma, \Gamma'] \Rightarrow_c \bot}{s : Y <: [\Gamma] \vdash s : Y <: [\Gamma]} \quad (\langle \_, \rangle_e) \\
\frac{\delta : Y \vdash \delta : Y}{\delta : Y, E : Y \Rightarrow_c T \vdash \delta[E] : \Rightarrow_c T} \quad (\Rightarrow_e) \\
\frac{s : Y <: [\Gamma] \vdash \delta : Y}{s : Y : Y \Rightarrow_c T \vdash \delta[s]_s : [\Gamma]} \quad (\Rightarrow_e) \\
\frac{s : Y <: [\Gamma], \delta : Y \vdash [c]_c Y \vdash s \delta (\forall \lambda y Y s \delta E_s) : [\Gamma] \Rightarrow_c \bot}{s : Y <: [\Gamma]} \quad (\forall)
\]

### 9. Stores

- Case \( \llbracket r x_i := t \rrbracket_{\tau} \). We only consider the case \( r (x_i := t) \), the proof for the case \( r (\alpha_i := E) \) is identical. This corresponds to the typing rules:

\[
\frac{\Gamma \vdash r : \Gamma' \quad \Gamma', r \vdash t : T}{\Gamma \vdash r (x_i := t) : \Gamma', x_i : T}
\]

By induction, we obtain two proofs of \( \vdash [r (x_i := t)]_{\tau} : [\Gamma] \Rightarrow_T \) and \( \vdash [r (x_i := t)]_{\tau} : [\Gamma, \Gamma'] \Rightarrow_T \). We can thus derive by Lemma F.1:

\[
\frac{\vdash [r (x_i := t)]_{\tau} : [\Gamma] \Rightarrow_T \vdash [r (x_i := t)]_{\tau} : [\Gamma, \Gamma'] \Rightarrow_T}{\vdash [r (x_i := t)]_{\tau} : [\Gamma, \Gamma'] \Rightarrow_T} \quad (r(t))
\]
A calculus of expandable stores  
Hugo Herbelin and Étienne Miquey

### G A typed call-by-name translation

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

\[ \begin{align*}
\langle t \mu x_n . c\rangle \tau &\rightarrow c[x_n/x] \tau [x_n := t] \quad \text{with } |\tau| = n \\
\langle \mu x_n . c \rangle E \tau &\rightarrow c[\alpha_n/\alpha] \tau[\alpha_n := E] \quad \text{with } |\tau| = n \\
\langle x_n E \rangle \tau &\rightarrow \langle \tau(n) E \rangle \tau \\
\langle V \alpha_n \rangle \tau &\rightarrow \langle V \tau(n) \rangle \tau \\
\langle \lambda x_n . t \rangle u \cdot E \tau &\rightarrow \langle u \mu x_n . (\langle t E \rangle \tau) \rangle \tau
\end{align*} \]

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

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

1. If $\Gamma \vdash V : T$ then $[\Gamma V : T]$.
2. If $\Gamma \vdash E : T^1$ then $[\Gamma E : T^2]$.
3. If $\Gamma \vdash t : T$ then $[\Gamma t : T]$.
4. If $\Gamma \vdash \epsilon : T^2$ then $[\Gamma \epsilon : T^3]$.
5. If $\Gamma \vdash c$ then $[\Gamma c]$.
6. If $\Gamma \vdash \epsilon$ then $[\Gamma \epsilon]$.
7. If $\Gamma \vdash \tau$ then $[\Gamma \tau]$.

**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 $[t]$, 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 \( x : A, y : B \vdash u : C \), through the translation we will thus have \( \Gamma \vdash [u]_1 : A, B \vdash C \rightarrow D \). 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 \( \tau_i : \tilde{U}_i \)) before returning \( V_i \) to its continuation. These terms can thus be assigned the types \( (A \rightarrow D) \rightarrow D, (B \rightarrow D) \rightarrow D, (C \rightarrow D) \rightarrow D \), and the closed term \( t_u \defeq t_0(\lambda x. t_1(\lambda y. t_2 u)) \) can thus be typed by \( \Gamma \vdash t_u : D \). 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 \( \tilde{U}_0, A, \tilde{U}_1, B, \tilde{U}_2 \).
A calculus of expandable stores

Hugo Herbelin and Étienne Miquey

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

1. If $\Gamma \vdash_V V : T$ then $\lfloor \Gamma \vdash_V V : T \rfloor$
2. If $\Gamma \vdash e : T$ then $\lfloor \Gamma \vdash e : T \rfloor$
3. If $\Gamma \vdash t : T$ then $\lfloor \Gamma \vdash t : T \rfloor$

\[\begin{align*}
\text{( Afterwards, the } n, 0, e, c \text{ in the source of the translation are analyzed, then) }
\end{align*}\]

\[\begin{align*}
\text{Values } V &:= \lambda x_i : t \mid x_i \mid k \\
\text{Terms } t, u &:= V \mid \mu \alpha_i : c \\
\text{Co-values } E &:= t \cdot e \mid \alpha_i \mid k \\
\text{Contexts } e &:= E \mid \mu x_i : c \\
\text{Environment } \tau &:= e \mid \tau[x_i := V] \mid \tau[\alpha_i := E] \\
\text{Commands } c &:= (t \cdot e) \\
\text{Closures } l &:= cr
\end{align*}\]

while the reduction rules are given by:

(Catch) \( \mu x_i : c \| e \| \tau \rightarrow c[\alpha_n / \alpha_i]r[\alpha_n := e] \) with $|r| = n$

(Let) \( V \| \mu x_i : c \| \tau \rightarrow c[x_n / x_i]r[x_n := V] \) with $|r| = n$

(Lookup) \( (V \| \alpha_n) \| \tau \rightarrow (V \| \tau(n)) \| \tau \)

(Beta) \( (\lambda x_i : t \cdot u \cdot e) \| \tau \rightarrow (u \| \mu x_i \cdot \tau(n)) \| \tau \)

The main specificity of the translations is that since only values can be stored in environments, we define $\mu \vdash T = V \| V$. The translations of types and judgments are similar to the translations 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 $e$ are analyzed, then context at level $e$ and finally values, the translation follows the same hierarchy. Again, we have:

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