6- Normalization of classical call-by-need

The call-by-need evaluation strategy

A famous functional programmer once was asked to give an overview talk. He began with : "This talk is about lazy functional programming and call by need." and paused. Then, quizzically looking at the audience, he quipped: "Are there any questions?" There were some, and so he continued: "Now listen very carefully, I shall say this only once."

This story, borrowed from [37], illustrates demand-driven computation and memoization of intermediate results, two key features of the call-by-need evaluation strategy that distinguish it from the call-by-name and call-by-value evaluation strategies (see Section 2.1.4).

The *call-by-name* evaluation strategy passes arguments to functions without evaluating them, postponing their evaluation to each place where the argument is needed, re-evaluating the argument several times if needed. For instance, the following reduction paths correspond to call-by-name evaluations in the λ-calculus extended with natural numbers:

$$\begin{array}{lllll}
(\lambda xy.yx)\,(2+3)\,(\lambda x.1) & \longrightarrow_\beta (\lambda y.y(2+3))\,\lambda x.1 & \longrightarrow_\beta (\lambda x.1)\,(2+3) & \longrightarrow_\beta 1 & \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x) & \longrightarrow_\beta (\lambda y.y(2+3))\,\lambda x.x & \longrightarrow_\beta (\lambda x.x)\,(2+3) & \longrightarrow_\beta 2+3 & \longrightarrow_\beta 5 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x\times x) & \xrightarrow{2}_\beta (\lambda x.x\times x)\,(2+3) & \longrightarrow_\beta (2+3)\times(2+3) & \xrightarrow{2}_\beta 5\times 5 & \longrightarrow_\beta 25
\end{array}$$

We observe for instance that (2 + 3) is never evaluated in the first example, while it is computed twice for the third one.

Conversely, the *call-by-value* evaluation strategy evaluates the arguments of a function into so-called "values" prior to passing them to the function. The evaluation is then shared between the different places where the argument is needed. Yet, if the argument is not needed, it is evaluated uselessly. The evaluation of the same examples in call-by-value gives:

$$\begin{array}{lllll}
(\lambda xy.yx)\,(2+3)\,(\lambda x.1) & \longrightarrow_\beta (\lambda xy.yx)\,5\,(\lambda x.1) & \longrightarrow_\beta (\lambda y.y5)\,(\lambda x.1) & \longrightarrow_\beta (\lambda x.1)\,5 & \longrightarrow_\beta 1 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x) & \longrightarrow_\beta (\lambda xy.yx)\,5\,(\lambda x.x) & \longrightarrow_\beta (\lambda y.y5)\,(\lambda x.x) & \longrightarrow_\beta (\lambda x.x)\,5 & \longrightarrow_\beta 5 \\
(\lambda xy.yx)\,(2+3)\,(\lambda x.x\times x) & \longrightarrow_\beta (\lambda xy.yx)\,5\,(\lambda x.x\times x) & \longrightarrow_\beta (\lambda y.y5)\,(\lambda x.x\times x) & \xrightarrow{2}_\beta 5\times 5 & \longrightarrow_\beta 25
\end{array}$$

We notice that in the first case, (2 + 3) is always evaluated once, which is better in the third case but useless in the first one. Also, remark that at the time where it is evaluated (the first step), it is impossible to predict how many times the argument will be used because it depends on the function that will be bind later to y (compare the second and third examples).

The *call-by-need* evaluation strategy is an evaluation strategy which evaluates arguments of functions only when needed, and, when needed, shares the computed results across all places where the argument is needed. In the first presentations of call-by-need λ-calculi [7, 112], this was done thanks to an additional $\texttt{let}\,x = \ldots\,\texttt{in}\ldots$ constructor. The first example, in call-by-need, reduces as follows:

$$\begin{array}{ll}
(\lambda xy.yx)\,(2+3)\,(\lambda x.1) & \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,(\lambda y.yx)\,(\lambda x.1) \\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = \lambda x.1\,\texttt{in}\,y\,x \\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = \lambda x.1\,\texttt{in}\,(\lambda x.1)x \\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = \lambda x.1\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,1
\end{array}$$

In particular, we observe that since it is never needed, $(2+3)$ is not evaluated. As for the third example, the reduction path is as follows[1]:

$$\begin{array}{rl}
(\lambda xy.yx)\,(2+3)\,(\lambda x.x\times x) & \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}(\lambda y.yx)\,(\lambda x.x\times x)\\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,yx\\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}(\lambda x.x\times x)x\\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,z\times z\\
& \longrightarrow_\beta \texttt{let}\,x = 2+3\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,x\times z\\
& \longrightarrow_\beta \texttt{let}\,x = 5\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,x\times z\\
& \longrightarrow_\beta \texttt{let}\,x = 5\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,5\times z\\
& \longrightarrow_\beta \texttt{let}\,x = 5\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,5\times x\\
& \longrightarrow_\beta \texttt{let}\,x = 5\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,5\times 5\\
& \longrightarrow_\beta \texttt{let}\,x = 5\,\texttt{in}\,\texttt{let}\,y = (\lambda x.x\times x)\,\texttt{in}\,\texttt{let}\,z = x\,\texttt{in}\,25
\end{array}$$

We see that each time that function is applied to an argument, the latter is lazily stored. When, further in the execution, $(2 + 3)$ is demanded by the left-member of the multiplication, its value is computed. Thanks to the $\texttt{let}\,x = \ldots\,\texttt{in}\ldots$ binder, this value is shared and when it is required a second time by the right-member of the multiplication, it is already available.

The call-by-need evaluation is at the heart of a functional programming language such as Haskell. It has in common with the call-by-value evaluation strategy that all places where a same argument is used share the same value. Nevertheless, it observationally behaves like the call-by-name evaluation strategy, in the sense that a given computation eventually evaluates to a value if and only if it evaluates to the same value (up to inner reduction) along the call-by-name evaluation. In particular, in a setting with non-terminating computations, it is not observationally equivalent to the call-by-value evaluation. Indeed, if the evaluation of a useless argument loops in the call-by-value evaluation, the whole computation loops (*e.g.* in $(\lambda\_.I)\,\Omega)$), which is not the case of call-by-name and call-by-need evaluations.

Continuation-passing style semantics

The call-by-name, call-by-value and call-by-need evaluation strategies can be turned into equational theories. For call-by-name and call-by-value, this was done by Plotkin [139] through continuation-passing style semantics characterizing these theories. For call-by-name, the corresponding induced equational theory[2] is Church's original theory of the λ-calculus based on the operational rule β.

For call-by-value, Plotkin showed that the induced equational theory includes the key operational rule β_V. The induced equational theory was further completed implicitly by Moggi [124] with the convenient introduction of a native `let` operator. Moggi's theory was then explicitly shown complete for CPS semantics by Sabry and Felleisen [148].

For the call-by-need evaluation strategy, a specific equational theory reflecting the intensional behavior of the strategy into a semantics was proposed independently by Ariola and Felleisen [3] and by Maraist, Odersky and Wadler [113]. A continuation-passing style semantics was proposed in the 90s by Okasaki, Lee and Tarditi [128]. However, this semantics does not ensure normalization of simply-typed call-by-need evaluation, as shown in [4], thus failing to ensure a property which holds in the simply-typed call-by-name and call-by-value cases (see Chapter 4).

[1]Observe that, as in the first example, we need to perform α-conversion on the fly, due to the $\texttt{let}\cdots = \ldots\,\texttt{in}\ldots$ bindings which behave like an explicit substitution. We will come back to this point in Section 6.4.1.

[2]Later on, Lafont, Reus and Streicher [103] gave a more refined continuation-passing style semantics which also validates the extensional rule η.

The $\overline{\lambda}_{lv}$-calculus: call-by-need with control

Continuation-passing style semantics *de facto* gives a semantics to the extension of λ-calculus with control operators, i.e. with operators such as Scheme's `call/cc`, Felleisen's $\mathcal{C}$, $\mathcal{K}$, or $\mathcal{A}$ operators [41], Parigot's μ and [ ] operators [130], Crolard's `catch` and `throw` operators [31]. In particular, even though call-by-name and call-by-need are observationally equivalent in the pure λ-calculus, their different intentional behaviors induce different continuation-passing style semantics, leading to different observational behaviors when control operators are considered.

Nonetheless, the semantics of calculi with control can also be reconstructed from an analysis of the duality between programs and their evaluation contexts, and the duality between the `let` construct (which binds programs) and a control operator such as Parigot's μ (which binds evaluation contexts). As explained in Chapter 4, such an analysis can be done in the context of the $\lambda\mu\tilde{\mu}$-calculus [32, 68].

Such an analysis is done in [4] in a variant of the $\lambda\mu\tilde{\mu}$-calculus which includes co-constants ranged over by $\boldsymbol{\kappa}$. Recall from Section 4.2 that the syntax of the $\lambda\mu\tilde{\mu}$-calculus can be refined into the following subcategories of terms and contexts:

Terms	t	$::=$	$\mu\alpha.c \mid V$	**Contexts**	e	$::=$	$\tilde{\mu}x.c \mid E$
Values	V	$::=$	$a \mid \lambda x.t \mid \boldsymbol{k}$	**Co-values**	E	$::=$	$\alpha \mid t \cdot e \mid \boldsymbol{\kappa}$

to which we add constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$. Then, by presenting reduction rules parameterized over a set of terms $\mathcal{V}$ and a set of evaluation contexts $\mathcal{E}$:

$$\begin{array}{lcll}
\langle t \| \tilde{\mu}x.c\rangle & \rightarrow & c[t/x] & t \in \mathcal{V} \\
\langle \mu\alpha.c \| e\rangle & \rightarrow & c[e/\alpha] & e \in \mathcal{E} \\
\langle \lambda x.t \| u \cdot e\rangle & \rightarrow & \langle u \| \tilde{\mu}x.\langle t \| e\rangle\rangle &
\end{array}$$

the difference between call-by-name and call-by-value can be characterized by the definition of these sets: the call-by-name evaluation strategy amounts to the case where $\mathcal{V} \triangleq$ *Proofs* and $\mathcal{E} \triangleq$ *Co-values* while call-by-value dually corresponds to $\mathcal{V} \triangleq$ *Values* and $\mathcal{E} \triangleq$ *Contexts*.

As for the call-by-need case, intuitively, we would like to set $\mathcal{V} \triangleq$ *Values* (we only substitute evaluated terms of which we share the value) and $\mathcal{E} \triangleq$ *Co-values* (a term is only reduced if it is in front of a co-value). However, such a definition is clearly not enough since any command of the shape $\langle \mu\alpha.c \| \tilde{\mu}x.c'\rangle$ would be blocked. We thus need to understand how the computation is driven forward, that is to say when we need to reduce terms. We observed that contexts that are either a co-constant $\boldsymbol{\kappa}$ or an applicative context[3] $t \cdot E$ eagerly demand a value. Such contexts are called *forcing contexts*, and denoted by F. When a variable x is in front of a forcing context, that is in $\langle x \| F\rangle$, the variable x is said to be *needed* or *demanded*. This allows us to identify meta-contexts C which are nesting of commands of the form $\langle t \| e\rangle$ for which neither t is in $\mathcal{V}$ (meaning it is some $\mu\alpha.c$) nor e in $\mathcal{E}$ (meaning it is an instance of some $\tilde{\mu}x.c$ which is not a forcing context). These contexts, defined by the following grammar:

Meta-contexts	$C[\,]$	$::=$	$[\,] \mid \langle \mu\alpha.c \| \tilde{\mu}x.C[\,]\rangle$

are such that in a $\tilde{\mu}$-binding of the form $\tilde{\mu}x.C[\langle x \| F\rangle]$, x is needed and a value is thus expected. These contexts, called *demanding contexts* are evaluation contexts whose evaluation is blocked on the evaluation of x, therefore requiring the evaluation of what is bound to x. In this case, we say that the bound variable x has been *forced.*

All this suggests another refinement of the syntax, introducing a division between *weak* co-values (resp. *weak* values), also called *catchable* contexts (since they are the one caught by a $\mu\alpha$ binder), and *strong* co-values (resp. *strong* values), which are precisely the forcing contexts. In comparison, with

[3] There is a restriction on the form of applicative contexts: the general form $t \cdot e$ is not necessarily a valid application, since for example in $\langle \mu\alpha.c \| t \cdot \tilde{\mu}x\langle y \| \alpha\rangle\rangle$, the context $t \cdot \tilde{\mu}x\langle y \| \alpha\rangle$ forces the execution of c even though its value is not needed. Applicative contexts are thus considered of the restricted shape $t \cdot E$.

$$\frac{(x:A)\in\Gamma}{\Gamma\vdash x:A\mid\Delta}\,(x)\qquad\frac{\Gamma,x:A\vdash t:B\mid\Delta}{\Gamma\vdash\lambda x.t:A\to B\mid\Delta}\,(\to_r)\qquad\frac{c:(\Gamma\vdash\Delta,\alpha:A)}{\Gamma\vdash\mu\alpha.c:A\mid\Delta}\,(\mu)$$

$$\frac{(\alpha:A)\in\Delta}{\Gamma\mid\alpha:A\vdash\Delta}\,(\alpha)\qquad\frac{\Gamma\vdash t:A\mid\Delta\qquad\Gamma\mid E:B\vdash\Delta}{\Gamma\mid t\cdot E:A\to B\vdash\Delta}\,(\to_l)\qquad\frac{c:(\Gamma,x:A\vdash\Delta)}{\Gamma\mid\tilde\mu x.c:A\vdash\Delta}\,(\tilde\mu)$$

$$\frac{\Gamma\vdash t:A\mid\Delta\qquad\Gamma\mid e:A\vdash\Delta}{\langle t\|e\rangle:(\Gamma\vdash\Delta)}\,(\textsc{Cut})\qquad\frac{(\boldsymbol{\kappa}:A)\in\mathcal{S}}{\Gamma\mid\boldsymbol{\kappa}:A\vdash\Delta}\,(\boldsymbol{\kappa})\qquad\frac{(\boldsymbol{k}:X)\in\mathcal{S}}{\Gamma\vdash\boldsymbol{k}:X\mid\Delta}\,(\boldsymbol{k})$$

Figure 6.1: Typing rules for $\overline{\lambda}_{lv}$

our former division, note that *catchable contexts* correspond to the union of former *co-values* with *demanding contexts*. Formally, the syntax is defined by[4]:

Strong values	v	$::=$	$\lambda x.t \mid \boldsymbol{k}$	**Forcing contexts**	F	$::=$	$t\cdot E \mid \boldsymbol{\kappa}$
Weak values	V	$::=$	$v \mid x$	**Catchable contexts**	E	$::=$	$F \mid \alpha \mid \tilde\mu x.C[\langle x\|F\rangle]$
Terms	t	$::=$	$V \mid \mu\alpha.c$	**Contexts**	e	$::=$	$E \mid \tilde\mu x.c$

We can finally define $\mathcal{V} \triangleq$ *Weak values* and $\mathcal{E} \triangleq$ *Catchable contexts*. The so-defined call-by-need calculus is close to the calculus called $\overline{\lambda}_{lv}$ in Ariola *et al* [4][5].

The $\overline{\lambda}_{lv}$ reduction, written as $\to_{lv}$, denotes thus the compatible reflexive transitive closure of the rules:

$$\begin{array}{rcl}\langle V\|\tilde\mu x.c\rangle & \to_{lv} & c[V/x]\\ \langle\mu\alpha.c\|E\rangle & \to_{lv} & c[E/\alpha]\\ \langle\lambda x.t\|u\cdot E\rangle & \to_{lv} & \langle u\|\tilde\mu x.\langle t\|E\rangle\rangle\end{array}$$

Observe that the next reduction is not necessarily at the top of the command, but may be buried under several bound computations $\mu\alpha.c$. For instance, the command $\langle\mu\alpha.c\|\tilde\mu x_1.\langle x_1\|\tilde\mu x_2.\langle x_2\|F\rangle\rangle\rangle$, where x_1 is not needed, reduces to $\langle\mu\alpha.c\|\tilde\mu x_1.\langle x_1\|F\rangle\rangle$, which now demands x_1.

The $\overline{\lambda}_{lv}$-calculus can be equipped with a type system (see Figure 6.1) made of the usual rules of the classical sequent calculus [32], where we adopt the convention that constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$ come with a signature $\mathcal{S}$ which assigns them a type.

Realizability and CPS interpretations of classical call-by-need

In the cases of the call-by-name and call-by-value evaluation strategies, the approach based on the $\lambda\mu\tilde\mu$-calculus leads to continuation-passing style semantics (Sections 4.4.4 and 4.5.3) similar to the ones given by Plotkin or, in the call-by-name case, also to the one by Lafont, Reus and Streicher [103]. In the case of call-by-need calculus, a continuation-passing style semantics for $\overline{\lambda}_{lv}$ is defined in [4] via a calculus called $\overline{\lambda}_{[lv\tau\star]}$. This calculus is equivalent to $\overline{\lambda}_{lv}$ but is presented in such a way that the head redex of a command can be found by looking only at the surface of the command, from which a continuation-passing style semantics directly comes. This semantics, distinct from the one in [128], is the object of study in this chapter.

The contribution of this chapter is twofold. On the one hand, we give a proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus. The normalization is obtained by means of a realizability interpretation of the calculus,

[4]In syntactic category, we implicitly assume $\tilde\mu x.c$ to only cover the cases which are not of the form $\tilde\mu x.C[\langle x\|F\rangle]$.

[5]The difference is in the fact that we had constants to preserve the duality. Also, a similar calculus, which we shall call weak $\overline{\lambda}_{lv}$, was previously studied in [6] with $\mathcal{E}$ defined instead to be $\tilde\mu x.C[\langle x\|E\rangle]$ (with same definition of C) and a definition of $\mathcal{V}$ which was different whether $\tilde\mu x.c$ was a forcing context ($\mathcal{V}$ was then the strong values) or not ($\mathcal{V}$ was then the weak values). Another variant is discussed in Section 6 of [4] where $\mathcal{E}$ is similarly defined to be $\tilde\mu x.C[\langle x\|E\rangle]$ and $\mathcal{V}$ is defined to be (uniformly) the strong values. All three semantics seem to make sense to us.

which is inspired from Krivine classical realizability [95]. As advocated in Section 4.3.3, the realizability interpretation is obtained by pushing one step further the methodology of Danvy's semantics artifacts already used in [4] to derive the continuation-passing-style semantics. While we only use it here to prove the normalization of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, our interpretation incidentally suggests a way to adapt Krivine's classical realizability to a call-by-need setting. This opens the door to the computational interpretation of classical proofs using lazy evaluation or shared memory cells.

On the other hand, we provide a type system for the continuation-passing-style transformation presented in [4] for the $\overline{\lambda}_{[lv\tau\star]}$-calculus such that the translation is well-typed. This presents various difficulties. First, since the evaluation of terms is shared, the continuation-passing-style translation is actually combined with a store-passing-style transformation. Second, as the store can grow along the execution, the translation also includes a Kripke-style forcing to address the extensibility of the store. This induces a target language which we call system F_Υ and which is an extension of Girard-Reynolds system F [60] and Cardelli system F$_{<:}$ [22]. Last but not least, the translation needs to take into account the problem of α-conversion. In a nutshell, this is due to the fact that terms can contain unbound variables that refer to elements of the store. So that a collision of names can result in auto-references and non-terminating terms. We deal with this in two-ways: we first elude the problem by using a fresh name generator and an explicit renaming of variables through the translation. Then we refine the translation to use De Bruijn levels to access elements of the store, which has the advantage of making it closer to an actual implementation. Surprisingly, the passage to De Bruijn levels also unveils some computational content related to the extension of stores.

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

6.1.1 Syntax

While all the results that are presented in the sequel of this chapter could be directly expressed using the $\overline{\lambda}_{lv}$-calculus, the continuation-passing style translation we present naturally arises from the decomposition of this calculus into a different calculus with an explicit *environment*, the $\overline{\lambda}_{[lv\tau\star]}$-calculus [4]. Indeed, as we shall explain thereafter, the decomposition highlights different syntactic categories that are deeply involved in the definition and the typing of the continuation-passing style translation.

The $\overline{\lambda}_{[lv\tau\star]}$-calculus is a reformulation of the $\overline{\lambda}_{lv}$-calculus with explicit environments, which we call *stores*, that are denoted by τ. Stores consists of a list of bindings of the shape $[x := t]$, where x is a term variable and t a term, and of bindings of the shape $[\alpha := e]$ where α is a context variable and e a context. For instance, in the closure $c\tau[x := t]\tau'$, the variable x is bound to t in c and τ'. Besides, the term t might be an unevaluated term (*i.e.* lazily stored), so that if x is eagerly demanded at some point during the execution of this closure, t will be reduced in order to obtain a value. In the case where t indeed produces a value V, the store will be updated with the binding $[x := V]$. However, a binding of this shape (with a value) is fixed for the rest of the execution. As such, our so-called stores somewhat behave like lazy explicit substitutions or mutable environments [6].

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

[6]To draw the comparison between our structures and the usual notions of stores and environments, two things should be observed. First, the usual notion of store refers to a structure of list that is fully mutable, in the sense that the cells can be updated at any time and thus values might be replaced. Second, the usual notion of environment designates a structure in which variables are bounded to closures made of a term and an environment. In particular, terms and environments are duplicated, *i.e.* sharing is not allowed. Such a structure resemble to a tree whose nodes are decorated by terms, as opposed to a machinery allowing sharing (like ours) whose the underlying structure is broadly a directed acyclic graphs. See for instance [104] for a Krivine abstract machine with sharing.

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

Figure 6.2: Reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

of the context $\tilde{\mu}[x].\langle x\|F\rangle\tau'$:

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

The reader can think of the previous rule as the "defrosting" operation of the frozen term $\mu\alpha.c$: this term is evaluated in the prefix of the store τ which predates it, in front of the context $\tilde{\mu}[x].\langle x\|F\rangle\tau'$ where the $\tilde{\mu}[x]$ binder is waiting for an (unfrozen) value. This context keeps trace of the suffix of the store τ' that was after the binding for x. This way, if a value V is indeed furnished for the binder $\tilde{\mu}[x]$, the original command $\langle x\|F\rangle$ is evaluated in the updated full store:

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

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

Note that our approach slightly differ from [4] in that we split values into two categories: strong values (v) and weak values (V). The strong values correspond to values strictly speaking. The weak values include the variables which force the evaluation of terms to which they refer into shared strong value. Their evaluation may require capturing a continuation. The syntax of the language is given by:

Strong values	v	$::=$	$\lambda x.t \mid \boldsymbol{k}$	**Forcing contexts**	F	$::=$	$\boldsymbol{\kappa} \mid t\cdot E$
Weak values	V	$::=$	$v \mid x$	**Catchable contexts**	E	$::=$	$F \mid \alpha \mid \tilde{\mu}[x].\langle x\|F\rangle\tau$
Terms	t	$::=$	$V \mid \mu\alpha.c$	**Evaluation contexts**	e	$::=$	$E \mid \tilde{\mu}x.c$

Closures	l	$::=$	$c\tau$
Commands	c	$::=$	$\langle t\|e\rangle$
Stores	τ	$::=$	$\varepsilon \mid \tau[x:=t] \mid \tau[\alpha:=E]$

The reduction, written $\to$, is the compatible reflexive transitive closure of the rules [7] given in Figure 6.2.

The different syntactic categories can be understood as the different levels of alternation in a context-free abstract machine: the priority is first given to contexts at level e (lazy storage of terms), then to terms at level t (evaluation of $\mu\alpha$ into values), then back to contexts at level E and so on until level v. These different categories are directly reflected in the definition of the context-free abstract machine (that we will present in Section 6.1.3) and in the continuation-passing style translation (and thus involved when typing it). We choose to highlight this by distinguishing different types of sequents already in the typing rules that we shall now present.

[7] We chose to make the substitutions of α variables effective while they are kept in an environment in [4]. This explains that we have one less rule.

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

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

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

$$
\frac{\Gamma, x : A, \Gamma' \vdash_F F : A^{\perp\!\!\!\perp} \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_E \tilde{\mu}[x].\langle x \| F \rangle \tau : A^{\perp\!\!\!\perp}}\,(\tilde{\mu}^{[]})
\qquad
\frac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_e e : A^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \| e \rangle}\,(c)
\qquad
\frac{\Gamma, \Gamma' \vdash_c c \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}\,(l)
$$

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

Figure 6.3: Typing rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.1.2 Type system

Unlike in the usual type system for sequent calculus where a judgment contains two typing contexts (one on the left for proofs, denoted by Γ, one on the right for contexts denoted by Δ), we use one-sided sequents (see Section 4.2.3.2): we group both typing contexts into one single context, denoting the types for contexts (that used to be in Δ) with the exponent $\perp\!\!\!\perp$. This allows us to draw a strong connection in the sequel between the typing context Γ and the store τ, which contain both kind of terms.

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

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

where types and typing contexts are defined by:

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

The typing rules are given on Figure 6.3 where we assume that a variable x (resp. co-variable α) only occurs once in a context Γ (we implicitly assume the possibility of renaming variables by α-conversion). This type system enjoys the property of subject reduction, whose proof is done by reasoning by induction over the derivation of the reduction $c\tau \to c'\tau'$, and relies on the fact that the type system admits a weakening rule.

Lemma 6.1. *The following rule is admissible for any level o of the hierarchy $e, t, E, V, F, v, c, l, \tau$:*

$$
\frac{\Gamma \vdash_o o : A \quad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_o o : A}\,(w)
$$

Proof. Easy induction on the structure of typing derivations obtained through the type system in Figure 6.3. □

Theorem 6.2 (Subject reduction). *If* $\Gamma \vdash_l c\tau$ *and* $c\tau \to c'\tau'$ *then* $\Gamma \vdash_l c'\tau'$.

Proof. By induction over the induction over the derivation of the reduction $c\tau \to c'\tau'$ (see Figure 6.2).

- **Case** $\langle t \| \tilde\mu x.c\rangle\tau \to c\tau[x := t]$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma' \vdash_t t : A} \quad \dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',x : A \vdash_c c}\,(c)}{\Gamma,\Gamma' \vdash_e \tilde\mu x.c : A}\,(\tilde\mu) \quad \dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'}}{\Gamma,\Gamma' \vdash_c \langle t \| \tilde\mu x.c\rangle}\,(c)}{\Gamma \vdash_l \langle t \| \tilde\mu x.c\rangle\tau}\,(l)$$

hence we can derive:

$$\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',x : A \vdash_c c}\,(c) \quad \dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'} \quad \dfrac{\Pi_t}{\Gamma,\Gamma' \vdash_t t : A}}{\Gamma \vdash_\tau \tau[x := t] : (\Gamma',x : A)}\,(\tau_t)}{\Gamma \vdash_l c\tau[x := t]}\,(l)$$

- **Case** $\langle \mu\alpha.c \| E\rangle\tau \to c\tau[\alpha := E]$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',\alpha : A^{\perp\!\!\!\perp} \vdash_c c}\,(c)}{\Gamma,\Gamma' \vdash_t \mu\alpha.c : A}\,(\mu) \quad \dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma' \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma' \vdash_e E : A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma' \vdash_c \langle \mu\alpha.c \| E\rangle}\,(c) \quad \dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'}}{\Gamma \vdash_l \langle \mu\alpha.c \| E\rangle\tau}\,(l)$$

hence we can derive:

$$\dfrac{\dfrac{\Pi_c}{\Gamma,\Gamma',\alpha : A^{\perp\!\!\!\perp} \vdash_c c}\,(c) \quad \dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash_\tau \tau : \Gamma'} \quad \dfrac{\Pi_E}{\Gamma,\Gamma' \vdash_E E : A}}{\Gamma \vdash_\tau \tau[\alpha := E] : (\Gamma',\alpha : A^{\perp\!\!\!\perp})}\,(\tau_E)}{\Gamma \vdash_l c\tau[\alpha := E]}\,(l)$$

- **Case** $\langle V \| \alpha\rangle\tau[\alpha := E]\tau' \to \langle V \| E\rangle\tau[\alpha := E]\tau'$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_t V : A} \quad \dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_F \alpha : A^{\perp\!\!\!\perp}}\,(\alpha)}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_E \alpha : A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_e \alpha : A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_c \langle V \| \alpha\rangle}\,(c) \quad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash \tau : \Gamma_0} \quad \dfrac{\Pi_E}{\Gamma,\Gamma_0 \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma \vdash_\tau \tau[\alpha := E] : \Gamma_0,\alpha : A^{\perp\!\!\!\perp}}\,(\tau_E) \quad \Pi_{\tau'}}{\Gamma \vdash_\tau \tau[\alpha := E]\tau' : \Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1}\,(\tau\tau')}{\Gamma \vdash_l \langle V \| \alpha\rangle\tau[\alpha := E]\tau'}\,(l)$$

where we cheated to compact each typing judgment for τ' (corresponding to types in Γ_1) in $\Pi_{\tau'}$. Therefore, we can derive:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_t V : A} \quad \dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_e E : A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1 \vdash_c \langle V \| E\rangle}\,(c) \quad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma \vdash \tau : \Gamma_0} \quad \dfrac{\Pi_E}{\Gamma,\Gamma_0 \vdash_E E : A^{\perp\!\!\!\perp}}}{\Gamma \vdash_\tau \tau[\alpha := E] : \Gamma_0,\alpha : A^{\perp\!\!\!\perp}}\,(\tau_E) \quad \Pi_{\tau'}}{\Gamma \vdash_\tau \tau[\alpha := E]\tau' : \Gamma_0,\alpha : A^{\perp\!\!\!\perp},\Gamma_1}\,(\tau\tau')}{\Gamma \vdash_l \langle V \| \alpha\rangle\tau[\alpha := E]\tau'}\,(l)$$

• **Case** $\langle x\|F\rangle\tau[x:=t]\tau' \to \langle t\|\tilde\mu[x].\langle x\|F\rangle\tau'\rangle\tau$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_V x:A}\,(x)}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_t x:A}\,(\uparrow^t) \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_c \langle x\|F\rangle}\,(c) \qquad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0} \quad \dfrac{\Pi_t}{\Gamma,\Gamma_0\vdash_t t:A}}{\Gamma\vdash_\tau \tau[x:=t]:\Gamma_0,x:A}\,(\tau_t) \quad \Pi_{\tau'}}{\Gamma\vdash_\tau \tau[x:=t]\tau':\Gamma_0,x:A,\Gamma_1}\,(\tau\tau')}{\Gamma\vdash_l \langle V\|F\rangle\tau[x:=t]\tau'}\,(l)$$

hence we can derive:

$$\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma_0,\Gamma_1\vdash_t t:A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_V x:A}\,(x)}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_t x:A}\,(\uparrow^t) \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_c \langle x\|F\rangle}\,(c) \qquad \dfrac{\Pi_{\tau'}}{\Gamma,\Gamma_0,x:A\vdash_\tau \tau':\Gamma_1}}{\Gamma,\Gamma_0,x:A\vdash_l \langle x\|F\rangle\tau'}\,(l)}{\Gamma,\Gamma_0\vdash_E \tilde\mu[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\tilde\mu^{[]})}{\Gamma,\Gamma_0\vdash_e \tilde\mu[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{}}{\Gamma,\Gamma_0\vdash_c \langle t\|\tilde\mu[x].\langle x\|F\rangle\tau'\rangle}\,(c) \qquad \dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0}}{\Gamma\vdash_l \langle t\|\tilde\mu[x].\langle x\|F\rangle\tau'\rangle\tau}\,(l)$$

• **Case** $\langle V\|\tilde\mu[x].\langle x\|F\rangle\tau'\rangle\tau \to \langle V\|F\rangle\tau[x:=V]\tau'$. A typing derivation of the closure on the left-hand side has the form:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,\Gamma_1\vdash_t V:A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_V x:A}\,(x)}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_t x:A}\,(\uparrow^t) \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_c \langle x\|F\rangle}\,(c) \qquad \dfrac{\Pi_{\tau'}}{\Gamma,\Gamma_0,x:A\vdash_\tau \tau':\Gamma_1}}{\Gamma,\Gamma_0,x:A\vdash_l \langle x\|F\rangle\tau'}\,(l)}{\Gamma,\Gamma_0\vdash_E \tilde\mu[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\tilde\mu^{[]})}{\Gamma,\Gamma_0\vdash_e \tilde\mu[x].\langle x\|F\rangle\tau':A^{\perp\!\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma_0\vdash_c \langle V\|\tilde\mu[x].\langle x\|F\rangle\tau'\rangle}\,(c) \qquad \dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0}}{\Gamma\vdash_l \langle V\|\tilde\mu[x].\langle x\|F\rangle\tau'\rangle\tau}\,(l)$$

Therefore we can derive:

$$\dfrac{\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_t V:A} \qquad \dfrac{\Pi_F}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_e F:A^{\perp\!\!\!\perp}}}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_c \langle V\|F\rangle}\,(c) \qquad \dfrac{\dfrac{\dfrac{\Pi_\tau}{\Gamma\vdash\tau:\Gamma_0} \quad \dfrac{\Pi_V}{\Gamma,\Gamma_0\vdash_t V:A}}{\Gamma\vdash_\tau \tau[x:=V]:\Gamma_0,x:A}\,(\tau_t) \quad \Pi_{\tau'}}{\Gamma\vdash_\tau \tau[x:=V]\tau':\Gamma_0,x:A,\Gamma_1}\,(\tau\tau')}{\Gamma\vdash_l \langle V\|F\rangle\tau[x:=V]\tau'}\,(l)$$

where we implicitly used Lemma 6.1 to weaken Π_V:

$$\dfrac{\dfrac{\Pi_V}{\Gamma,\Gamma_0\vdash_t V:A} \quad \Gamma,\Gamma_0\subseteq\Gamma,\Gamma_0,x:A,\Gamma_1}{\Gamma,\Gamma_0,x:A,\Gamma_1\vdash_t V:A}\,(w)$$

• **Case** $\langle \lambda x.t \| u \cdot E\rangle\tau \rightarrow \langle u \| \tilde{\mu}x.\langle t\|E\rangle\rangle\tau$. A typing proof for the closure on the left-hand side is of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma',x:A\vdash_t t:B}}{\Gamma,\Gamma'\vdash_v \lambda x.t:A\to B}\,(\to_r)}{\Gamma,\Gamma'\vdash_V \lambda x.t:A\to B}\,(\uparrow^V)}{\Gamma,\Gamma'\vdash_t \lambda x.t:A\to B}\,(\uparrow^t) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Pi_u}{\Gamma,\Gamma'\vdash_t u:A} \quad \dfrac{\Pi_E}{\Gamma,\Gamma'\vdash_E E:B^{\perp\!\!\perp}}}{\Gamma,\Gamma'\vdash_F u\cdot E:(A\to B)^{\perp\!\!\perp}}\,(\to_l)}{\Gamma,\Gamma'\vdash_E u\cdot E:(A\to B)^{\perp\!\!\perp}}\,(\uparrow^E)}{\Gamma,\Gamma'\vdash_e u\cdot E:(A\to B)^{\perp\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma'\vdash_c \langle\lambda x.t\|u\cdot E\rangle}\,(c) \qquad \dfrac{\Pi_\tau}{\Gamma\vdash_\tau \tau:\Gamma'}}{\Gamma\vdash_l \langle\lambda x.t\|u\cdot E\rangle\tau}\,(l)$$

We can thus build the following derivation:

$$\dfrac{\dfrac{\dfrac{\Pi_u}{\Gamma,\Gamma'\vdash_t u:A} \qquad \dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma,\Gamma',x:A\vdash_t t:B} \quad \dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma',x:A\vdash_E E:B^{\perp\!\!\perp}}}{\Gamma,\Gamma',x:A\vdash_e E:B^{\perp\!\!\perp}}\,(\uparrow^e)}{\Gamma,\Gamma',x:A\vdash_c \langle t\|E\rangle}\,(c)}{\Gamma,\Gamma'\vdash_e \tilde{\mu}x.\langle t\|E\rangle:A^{\perp\!\!\perp}}\,(\tilde{\mu})}{\Gamma,\Gamma'\vdash_c \langle u\|\tilde{\mu}x.\langle t\|E\rangle\rangle}\,(c) \qquad \dfrac{\Pi_\tau}{\Gamma\vdash_\tau \tau:\Gamma'}}{\Gamma\vdash_l \langle u\|\tilde{\mu}x.\langle t\|E\rangle\rangle\tau}\,(l)$$

where we implicitly used Lemma 6.1 to weaken Π_E:

$$\dfrac{\dfrac{\Pi_E}{\Gamma,\Gamma\vdash_E E:B^{\perp\!\!\perp}} \quad \Gamma,\Gamma'\subseteq\Gamma,\Gamma',x:A}{\Gamma,\Gamma',x:A\vdash_E E:B^{\perp\!\!\perp}}\,(w)$$

□

6.1.3 Small-step reductions rules

As in the cases of the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi (see Sections 4.4 and 4.5), the reduction system can be decomposed into small-step reduction rules. We annotate again commands with the level of syntax we are examining ($c_e, c_t, \dots$), and define a new set of reduction rules which separate computational steps (corresponding to big-step reductions), and administrative steps, which organize the descent in the syntax. In order, a command first put the focus on the context at level e, then on the term at level t, and so on following the hierarchy e, t, E, V, F, v. This results again in an abstract machine in context-free form, since each step only analyzes one component of the command, the "active" term or context, and is parametric in the other "passive" component. In essence, for each phase of the machine, either the term or the context is fully in control and independent, regardless of what the other half happens to be.

We recall the resulting abstract machine from [4] in Figure 6.4. Except for a subtlety of α-conversion that we will explain in Section 6.4.1, these rules directly lead to the definition of the CPS in [4] that we shall type in the next sections. Furthermore, the realizability interpretation *à la* Krivine (that we are about to present in the coming section) is deeply based upon this set of rules. Indeed, remember that a realizer is precisely a term which is going to behave well in front of any opponent in the opposed falsity value. We shall thus take advantage of the context-free rules where at each level, the reduction step is defined independently of the passive component.

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

Figure 6.4: Context-free abstract machine for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.2 Realizability interpretation of the simply-typed $\overline{\lambda}_{[lv\tau\star]}$-calculus

6.2.1 Normalization by realizability

The proof of normalization for the $\overline{\lambda}_{[lv\tau\star]}$-calculus that we present in this section is inspired from techniques of Krivine's classical realizability [95], whose notations we borrow. Actually, it is also very close to a proof by reducibility[8]. In a nutshell, to each type A is associated a set $|A|_t$ of terms whose execution is guided by the structure of A. These terms are the ones usually called *realizers* in Krivine's classical realizability. Their definition is in fact indirect, and is done by orthogonality to a set of "correct" computations, called a *pole*. The choice of this set is central when studying models induced by classical realizability for second-order-logic, but in the present case we only pay attention to the particular pole of terminating computations. This is where lies the main difference with a proof by reducibility, where everything is done with respect to SN, while our definition are parametric in the pole (which is chosen to be the set of normalizing closures in the end). The adequacy lemma, which is the central piece, consists in proving that typed terms belong to the corresponding sets of realizers, and are thus normalizing.

More in details, our proof can be sketched as follows. First, we generalize the usual notion of closed term to the notion of closed *term-in-store*. Intuitively, this is due to the fact that we are no longer interested in closed terms and substitutions to close open terms, but rather in terms that are closed when considered in the current store. This is based on the simple observation that a store is nothing more than a shared substitution whose content might evolve along the execution. Second, we define the notion of *pole* $\perp\!\!\!\perp$, which are sets of closures closed by anti-evaluation and store extension. In particular, the set of normalizing closures is a valid pole. This allows us to relate terms and contexts thanks to a notion of orthogonality with respect to the pole. We then define for each formula A and typing level o (of e, t, E, V, F, v) a set $|A|_o$ (resp. $\|A\|_o$) of terms (resp. contexts) in the corresponding syntactic category. These sets correspond to reducibility candidates, or to what is usually called truth values and falsity values in realizability.

Finally, the core of the proof consists in the adequacy lemma, which shows that any closed term of type A at level o is in the corresponding set $|A|_o$. This guarantees that any typed closure is in any pole, and in particular in the pole of normalizing closures. Technically, the proof of adequacy evaluates in each case a state of an abstract machine (in our case a closure), so that the proof also proceeds by

[8]See for instance the proof of normalization for system D presented in [92, 3.2])

evaluation. A more detailed explanation of this observation as well as a more introductory presentation of normalization proofs by classical realizability are given in an article by Dagand and Scherer [35].

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

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

Definition 6.3 (Closed store). We extend the notion of free variable to stores:

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

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

Definition 6.4 (Compatible stores). We say that two stores τ and τ' are *independent* and note $\tau\#\tau'$ when $\mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') = \emptyset$. We say that they are *compatible* and note $\tau \diamond \tau'$ whenever for all variables x (resp. co-variables α) present in both stores: $x \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')$; the corresponding terms (resp. contexts) in τ and τ' coincide: formally $\tau = \tau_0[x := t]\tau_1$ and $\tau' = \tau_0'[x := t]\tau_1'$. Finally, we say that τ' is an *extension* of τ and note $\tau \lhd \tau'$ whenever $\mathrm{dom}(\tau) \subseteq \mathrm{dom}(\tau')$ and $\tau \diamond \tau'$. ⌟

Definition 6.5 (Compatible union). We denote by $\overline{\tau\tau'}$ the compatible union $\mathtt{join}(\tau,\tau')$ of closed stores τ and τ', defined by:

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

⌟

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

Lemma 6.6. *If τ and τ' are two compatible stores, then $\tau \lhd \overline{\tau\tau'}$ and $\tau' \lhd \overline{\tau\tau'}$. Besides, if τ is of the form $\tau_0[x := t]\tau_1$, then $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x := t]\overline{\tau_1}$ with $\tau_0 \lhd \overline{\tau_0}$ and $\tau_1 \lhd \overline{\tau_1}$.*

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

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

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

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

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

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

Definition 6.7 (Term-in-store)**.** We call *closed term-in-store* (resp. *closed context-in-store*, *closed closures*) the combination of a term t (resp. context e, command c) with a closed store τ such that $FV(t) \subseteq \operatorname{dom}(\tau)$. We use the notation $(t|\tau)$ to denote such a pair. ⌟

We should note that in particular, if t is a closed term, then $(t|\tau)$ is a term-in-store for any closed store τ. The notion of closed term-in-store is thus a generalization of the notion of closed terms, and we will (ab)use of this terminology in the sequel. We denote the sets of closed closures by C_0, and will identify $(c|\tau)$ and the closure $c\tau$ when c is closed in τ. Observe that if $c\tau$ is a closure in C_0 and τ' is a store extending τ, then $c\tau'$ is also in C_0. We are now equipped to define the notion of pole, and verify that the set of normalizing closures is indeed a valid pole.

Definition 6.8 (Pole)**.** A subset $\perp\!\!\!\perp \subseteq C_0$ is said to be *saturated* or *closed by anti-reduction* whenever for all $(c|\tau), (c'|\tau') \in C_0$, if $c'\tau' \in \perp\!\!\!\perp$ and $c\tau \to c'\tau'$ then $c\tau \in \perp\!\!\!\perp$. It is said to be *closed by store extension* if whenever $c\tau \in \perp\!\!\!\perp$, for any store τ' extending τ: $\tau \lhd \tau'$, $c\tau' \in \perp\!\!\!\perp$. A *pole* is defined as any subset of C_0 that is closed by anti-reduction and store extension. ⌟

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

Proposition 6.9. *The set* $\perp\!\!\!\perp_{\Downarrow} = \{c\tau \in C_0 : \ c\tau \textit{ normalizes}\ \}$ *is a pole.*

Proof. As we only considered closures in C_0, both conditions (closure by anti-reduction and store extension) are clearly satisfied:

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

Definition 6.10 (Orthogonality)**.** Given a pole $\perp\!\!\!\perp$, we say that a term-in-store $(t|\tau)$ is *orthogonal* to a context-in-store $(e|\tau')$ and write $(t|\tau)\perp\!\!\!\perp(e|\tau')$ if τ and τ' are compatible and $\langle t\|e\rangle\overline{\tau\tau'} \in \perp\!\!\!\perp$. ⌟

Remark 6.11. The reader familiar with Krivine's forcing machine [98] might recognize his definition of orthogonality between terms of the shape (t,p) and stacks of the shape (π,q), where p and q are forcing conditions:

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

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

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

Definition 6.12 (Realizers)**.** Given a fixed pole $\perp\!\!\!\perp$, we set:

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

⌟

Remark 6.13. We draw the reader attention to the fact that we should actually write $|A|_v^{\perp\!\!\!\perp}$, $\|A\|_F^{\perp\!\!\!\perp}$, etc... and $\tau \Vdash_{\perp\!\!\!\perp} \Gamma$, because the corresponding definitions are parameterized by a pole $\perp\!\!\!\perp$. As it is common in Krivine's classical realizability, we ease the notations by removing the annotation $\perp\!\!\!\perp$ whenever there is no ambiguity on the pole. ⌟

If the definition of the different sets might seem complex at first sight, we claim that they are quite natural with regard to the methodology of Danvy's semantics artifacts presented in [4]. Indeed, having an abstract machine in context-free form (the last step in this methodology before deriving the CPS) allows us to have both the term and the context (in a command) that behave independently of each other. Intuitively, a realizer at a given level is precisely a term which is going to behave well (be in the pole) in front of any opponent chosen in the previous level (in the hierarchy v, F, V,etc...). For instance, in a call-by-value setting, there are only three levels of definition (values, contexts and terms) in the interpretation, because the abstract machine in context-free form also has three. Here the ground level corresponds to strong values, and the other levels are somewhat defined as terms (or context) which are well-behaved in front of any opponent in the previous one. The definition of the different sets $|A|_v, \|A\|_F, |A|_V$, etc... directly stems from this intuition.

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

Proposition 6.14. *For any type A and any given pole $\perp\!\!\!\perp$, we have the following inclusions:*

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

Proof. All the inclusions are proved in a similar way. We only give the proof for $|A|_v \subseteq |A|_V$. Let $\perp\!\!\!\perp$ be a pole and $(v|\tau)$ be in $|A|_v$. We want to show that $(v|\tau)$ is in $|A|_V$, that is to say that v is in the syntactic category V (which is true), and that for any $(F|\tau') \in \|A\|_F$ such that $\tau \diamond \tau'$, $(v|\tau) \perp\!\!\!\perp (F|\tau')$. The latter holds by definition of $(F|\tau') \in \|A\|_F$, since $(v|\tau) \in |A|_v$. □

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

Definition 6.15. Given a closed store τ and a fixed pole $\perp\!\!\!\perp$, we say that τ *realizes* Γ, which we write[9] $\tau \Vdash \Gamma$, if:

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

⌟

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

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

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

Lemma 6.16 (Store weakening). *Let τ and τ' be two stores such that $\tau \lhd \tau'$, let Γ be a typing context and let $\perp\!\!\!\perp$ be a pole. The following statements hold:*

1. $\overline{\tau\tau'} = \tau'$

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

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

Proof. 1. Straightforward from the definitions.

2. This essentially amounts to the following observations. First, one remarks that if $(t|\tau)$ is a closed term, so is $(t|\overline{\tau\tau'})$ for any store τ' compatible with τ. Second, we observe that if we consider for instance a closed context $(E|\tau'') \in \|A\|_E$, then $\overline{\tau\tau'}\diamond\tau''$ implies $\tau\diamond\tau''$, thus $(t|\tau)\perp\!\!\!\perp(E|\tau'')$ and finally $(t|\overline{\tau\tau'})\perp\!\!\!\perp(E|\tau'')$ by closure of the pole under store extension. We conclude that $(t|\tau')\perp\!\!\!\perp(E|\tau'')$ using the first statement.
3. By definition, for all $(x : A) \in \Gamma$, τ is of the form $\tau_0[x := t]\tau_1$ such that $(t|\tau_0) \in |A|_t$. As τ and τ' are compatible, we know by Lemma 8.16 that $\overline{\tau\tau'}$ is of the form $\tau_0'[x := t]\tau_1'$ with τ_0' an extension of τ_0, and using the first point we get that $(t|\tau_0') \in |A|_t$. □

We are now equipped to prove the adequacy of the type system for the $\overline{\lambda}_{[l\upsilon\tau\star]}$-calculus with respect to the realizability interpretation.

Definition 6.17 (Adequacy)**.** Given a fixed pole $\perp\!\!\!\perp$, we say that:

- A typing judgment $\Gamma \vdash_t t : A$ is *adequate* (w.r.t. the pole $\perp\!\!\!\perp$) if for all stores $\tau \Vdash \Gamma$, we have $(t|\tau) \in |A|_t$.
- More generally, we say that an inference rule

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

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

⌟

Remark 6.18. 1. As usual, it is clear from the latter definition that a typing judgment that is derivable from a set of adequate inference rules is adequate too.

2. The interpretation we gave here relies on the fact that the calculus is simply-typed with constants inhabiting the atomic types. If we were interested in open formulas (or second-order logic), we should as usual (see Section 3.4.4) consider valuation to close formulas, which would map second-order variables to set of strong values. ⌟

Proposition 6.19 (Adequacy)**.** *The typing rules of Figure 6.3 for the* $\overline{\lambda}_{[l\upsilon\tau\star]}$*-calculus without co-constants are adequate with any pole. In other words, if* Γ *is a typing context,* $\perp\!\!\!\perp$ *a pole and* τ *a store such that* $\tau \Vdash \Gamma$*, then the following holds:*

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

Proof. We proceed by induction over the typing rules.

- **Case** Constants. This case stems directly from the definition of $|X|_v$ for X atomic.

- **Case** $(\to_r)$. This case exactly matches the definition of $|A \to B|_v$. Assume that

$$\frac{\Gamma, x : A \vdash_t t : B}{\Gamma \vdash_v \lambda x.t : A \to B}\ (\to_r)$$

and let $⫫$ be a pole and τ a store such that $\tau \Vdash \Gamma$. If $(u|\tau')$ is a closed term in the set $|A|_t$, then, up to α-conversion for the variable x, $\overline{\tau\tau'} \Vdash \Gamma$ by Lemma 6.16 and $\overline{\tau\tau'}[x := u] \Vdash \Gamma, x : A$. Using the induction hypothesis, $(t|\overline{\tau\tau'}[x := u])$ is indeed in $|B|_t$.

- **Case** $(\to_l)$. Assume that

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

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

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

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

- **Case** $(\uparrow^V)$. This case, as well as every other case where typing a term (resp. context) at a higher level of the hierarchy (rules $(\uparrow^E)$, $(\uparrow^t)$, $(\uparrow^e)$), is a simple consequence of Proposition 6.14. Indeed, assume for instance that

$$\frac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}\ (\uparrow^V)$$

and let $⫫$ be a pole and τ a store such that $\tau \Vdash \Gamma$. By induction hypothesis, we get that $(v|\tau) \in |A|_v$. Thus, if $(F|\tau')$ is in $\|A\|_F$, by definition $(v|\tau) ⫫ (F|\tau')$.

- **Case** (x). Assume that

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

and let $⫫$ be a pole and τ a store such that $\tau \Vdash \Gamma$. As $(x : A) \in \Gamma$, we know that τ is of the form $\tau_0[x := t]\tau_1$ with $(t|\tau_0) \in |A|_t$. Let $(F|\tau')$ be in $\|A\|_F$, with $\tau \diamond \tau'$. By Lemma 8.16, we know that $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x := t]\overline{\tau_1}$. Hence, we have:

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

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

- **Case** $(\tilde{\mu}^{[]})$. Assume that

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

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

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

By induction hypothesis, we obtain $\tau[x := V]\tau' \Vdash \Gamma, x : A, \Gamma'$. Up to α-conversion in F and τ', so that the variables in τ' are disjoint from those in τ_0, we have that $\overline{\tau_0\tau} \Vdash \Gamma$ (by Lemma 6.16) and then $\tau'' \triangleq \overline{\tau_0\tau}[x := V]\tau' \Vdash \Gamma, x : A, \Gamma'$. By induction hypothesis again, we obtain that $(F|\tau'') \in \|A\|_F$ (this was an assumption in the previous case) and as $(V|\tau_0) \in |A|_V$, we finally get that $(V|\tau_0) \perp\!\!\!\perp (F|\tau'')$ and conclude again by anti-reduction.

- **Cases** (α). This case is obvious from the definition of $\tau \Vdash \Gamma$.

- **Case** (μ). Assume that

$$\frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash_c c}{\Gamma \vdash_t \mu\alpha.c : A}\ (\mu)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(E|\tau')$ be a closed context in $\|A\|_E$ such that $\tau \diamond \tau'$. We have that :

$$\langle \mu\alpha.c \| E\rangle\overline{\tau\tau'} \quad\to\quad c\overline{\tau\tau'}[\alpha := E]$$

Using the induction hypothesis, we only need to show that $\overline{\tau\tau'}[\alpha := E] \Vdash \Gamma, \alpha : A^{\perp\!\!\!\perp}, \Gamma'$ and conclude by anti-reduction. This obviously holds, since $(E|\tau') \in \|A\|_E$ and $\overline{\tau\tau'} \Vdash \Gamma$ by Lemma 8.16.

- **Case** ($\tilde{\mu}$). This case is identical to the previous one.

- **Case** (c). Assume that

$$\frac{\Gamma \vdash_t t : A \quad \Gamma \vdash_e e : A^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \| e\rangle}\ (c)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Then by induction hypothesis $(t|\tau) \in |A|_t$ and $(e|\tau) \in \|A\|_e$, so that $\langle t\|e\rangle\tau \in \perp\!\!\!\perp$.

- **Case** (τ_t). This case directly stems from the induction hypothesis which exactly matches the definition of $\tau\tau'[x := t] \Vdash \Gamma, \Gamma', x : A$. The case for the rule (τ_E) is identical, and the case for the rule (ε) is trivial.

- **Case** (l). This case is a direct consequence of induction hypotheses for τ and c. Assume indeed that:

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

Then by induction hypotheses $\tau\tau' \Vdash \Gamma, \Gamma'$ and thus $c\tau\tau' \in \perp\!\!\!\perp$.

□

The previous result required to consider the $\overline{\lambda}_{[l v\tau\star]}$-calculus without co-constants. Indeed, we consider co-constants as coming with their typing rules, potentially giving them any type (whereas constants can only be given an atomic type). Thus, there is *a priori* no reason[10] why their types should be adequate with any pole.

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

Corollary 6.20. *If $c\tau$ is a closure such that $\vdash_l c\tau$ is derivable, then for any pole $\perp\!\!\!\perp$ such that the typing rules for co-constants used in the derivation are adequate with $\perp\!\!\!\perp$, $c\tau \in \perp\!\!\!\perp$.*

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

We can now put our focus back on the normalization of typed closures. As we already saw in Proposition 6.9, the set $⫫_{\Downarrow}$ of normalizing closure is a valid pole, so that it only remains to prove that any typing rule for co-constants is adequate with $⫫_{\Downarrow}$.

Lemma 6.21. *Any typing rule for co-constants is adequate with the pole* $⫫_{\Downarrow}$, i.e. *if* Γ *is a typing context, and* τ *is a store such that* $\tau \Vdash \Gamma$, *if* $\boldsymbol{\kappa}$ *is a co-constant such that* $\Gamma \vdash_F \boldsymbol{\kappa} : A^{⫫}$, *then* $(\boldsymbol{\kappa}|\tau) \in \|A\|_F$.

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

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

Theorem 6.22. *If* $c\tau$ *is a closure of the* $\overline{\lambda}_{[lv\tau\star]}$*-calculus such that* $\vdash_l c\tau$ *is derivable, then* $c\tau$ *normalizes.*

Besides, the translations[11] from $\overline{\lambda}_{lv}$ to $\overline{\lambda}_{[lv\tau\star]}$ defined by Ariola *et al.* both preserve normalization of commands [4, Theorem 2,4]. As it is clear that they also preserve typing, the previous result also implies the normalization of the $\overline{\lambda}_{lv}$-calculus:

Corollary 6.23. *If* c *is a closure of the* $\overline{\lambda}_{lv}$*-calculus such that* $c : (\vdash)$ *is derivable, then* c *normalizes.*

This is to be contrasted with Okasaki, Lee and Tarditi's semantics for the call-by-need λ-calculus, which is not normalizing in the simply-typed case, as shown in Ariola *et al* [4].

6.3 A typed store-and-continuation-passing style translation

Guided by the normalization proof of the previous section, we shall now present a type system adapted to the continuation-passing style translation defined in [4]. The computational part is almost the same, except for the fact that we explicitly handle renaming through a substitution σ that replaces names of the source language by names of the target.

6.3.1 Guidelines of the translation

The transformation is actually not only a continuation-passing style translation. Because of the sharing of the evaluation of arguments, the store associating terms to variables has to be passed around. Passing the store amounts to combining the continuation-passing style translation with a store-passing style translation. Additionally, the store is extensible, so, to anticipate extension of the store, Kripke style forcing has to be used too, in a way comparable to what is done in step-indexing translations. Before presenting in detail the target system of the translation, let us explain step by step the rationale guiding the definition of the translation. To facilitate the comprehension of the different steps, we illustrate each of them with the translation of the sequent $a : A, \alpha : A^{⫫}, b : B \vdash_e e : C$.

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

As shown in [4] and as emphasized by the definition of realizers (see Definition 6.12) reflecting the 6 nested syntactic categories used to define $\overline{\lambda}_{[lv\tau\star]}$, there are 6 different levels of control in call-by-need, leading to 6 mutually defined levels of interpretation. We define $[\![A \to B]\!]_v$ for strong values as $[\![A]\!]_t \to [\![B]\!]_E$, we define $[\![A]\!]_F$ for forcing contexts as $\neg\, [\![A]\!]_v$, $[\![A]\!]_V$ for weak values as $\neg\, [\![A]\!]_F \overset{2}{=\neg}\, [\![A]\!]_v$, and so on until $[\![A]\!]_e$ defined as $\overset{5}{\neg}\, [\![A]\!]_v$ (where $\overset{0}{\neg}\, A \triangleq A$ and $\overset{n+1}{\neg}\, A \triangleq \neg \overset{n}{\neg}\, A$).

As we already observed in the previous section (see Definition 8.18), hypotheses from a context Γ of the form $\alpha : A^{⫫}$ are to be translated as $[\![A]\!]_E \overset{3}{=\neg}\, [\![A]\!]_v$ while hypotheses of the form $x : A$ are to be

[11]There is actually an intermediate step to a calculus named $\bar{\lambda}_{[l\tau v]}$.

translated as $[\![A]\!]_t \overset{4}{=\neg} [\![A]\!]_v$. Up to this point, if we denote this translation of Γ by $[\![\Gamma]\!]$, in the particular case of $\Gamma \vdash_t A$ the translation is $[\![\Gamma]\!] \vdash [\![A]\!]_t$ and similarly for other levels, *e.g.* $\Gamma \vdash_e A$ translates to $[\![\Gamma]\!] \vdash [\![A]\!]_e$.

Example 6.24 (Translation, step 1)**.** Up to now, the translation taking into account the continuation-passing style of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ is simply:

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

⌟

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

$$\begin{aligned}[\![\Gamma]\!] \triangleright_t A &= [\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_E A) \to \bot \\ &= [\![\Gamma]\!] \to ([\![\Gamma]\!] \to ([\![\Gamma]\!] \triangleright_V A) \to \bot) \to \bot = \ldots\end{aligned}$$

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

Example 6.25 (Translation, step 2)**.** Up to now, the continuation-and-store passing style translation of $a : A, \alpha : A^{\perp\!\!\!\perp}, b : B \vdash_e e : C$ is:

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

where:

$$\begin{aligned}[\![a : A, \alpha : A^{\perp\!\!\!\perp}, b : B]\!] &= [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \triangleright_t B \\ &= [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \to ([\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] \triangleright_E B) \to \bot = \ldots \\ [\![a : A, \alpha : A^{\perp\!\!\!\perp}]\!] &= [\![a : A]\!],\ \alpha : [\![a : A]\!] \triangleright_E A \\ &= [\![a : A]\!],\ \alpha : [\![a : A]\!] \to ([\![a : A]\!] \to \triangleright_E A) \to \bot = \ldots \\ [\![a : A]\!] &= a : \varepsilon \triangleright_t A \ = \ a :\overset{4}{\neg} [\![A]\!]_v\end{aligned}$$

⌟

Step 3 - Extension of the store. The store-passing style part being settled, it remains to anticipate that the store is extensible. This is done by supporting arbitrary insertions of any term at any place in the store. The extensibility is obtained by quantifying over all possible extensions of the store at each level of the negation. This corresponds to the intuition that in the realizability interpretation, given a sequent $\Gamma \vdash_t t : A$ we showed that for any store τ such that $\tau \Vdash \Gamma$, we had $(t|\tau)$ in $|A|_t$. But the definition of $\tau \Vdash \Gamma$ is such that for any $\Gamma' \supseteq \Gamma$, if $\tau \Vdash \Gamma'$ then $\tau \Vdash \Gamma$, so that actually $(t|\tau')$ is also $|A|_t$. The term t was thus compatible with any extension of the store.

For this purpose, we use as a type system an adaptation of System $F_{<:}$ [22] extended with stores, defined as lists of assignations $[x := t]$. *Store types*, denoted by Υ, are defined as list of types of the form $(x : A)$ where x is a name and A is a type properly speaking and admit a subtyping notion $\Upsilon' <: \Upsilon$ to express that Υ' is an extension of Υ. This corresponds to the following refinement of the definition of $[\![\Gamma]\!] \triangleright_t A$:

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

The reader can think of subtyping as a sort of Kripke forcing [89], where *worlds* are store types Υ and *accessible worlds* from Υ are precisely all the possible $\Upsilon' <: \Upsilon$.

Example 6.26 (Translation, step 3)**.** The translation, now taking into account store extensions, of $a : A, \alpha : A^{\perp\!\!\perp}, b : B \vdash_e e : C$ becomes:

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

where:

$$\begin{array}{rcl}
[\![a : A, \alpha : A^{\perp\!\!\perp}, b : B]\!] & = & [\![a : A, \alpha : A^{\perp\!\!\perp}]\!],\ b : [\![a : A, \alpha : A^{\perp\!\!\perp}]\!] \triangleright_t B \\
& = & [\![a : A, \alpha : A^{\perp\!\!\perp}]\!],\ b : \forall \Upsilon <: [\![a : A, \alpha : A^{\perp\!\!\perp}]\!].\Upsilon \to (\Upsilon \triangleright_E B) \to \bot = \ldots \\
[\![a : A, \alpha : A^{\perp\!\!\perp}]\!] & = & [\![a : A]\!],\ \alpha : [\![a : A]\!] \triangleright_E A \\
& = & [\![a : A]\!],\ \alpha : \forall \Upsilon <: [\![a : A]\!].\Upsilon \to (\Upsilon \to \triangleright_E A) \to \bot = \ldots \\
[\![a : A]\!] & = & a : \varepsilon \triangleright_t A \ = \ a : \forall \Upsilon.\Upsilon \to (\Upsilon \triangleright_E A) \to \bot
\end{array}$$

⌟

Step 4 - Explicit renaming As we will explain in details in the next section (see Section 6.4.1), we need to handle the problem of renaming the variables during the translation. We assume that we dispose of a generator of fresh names (in the target language). In practice, this means that the implementation of the CPS requires for instance to have a list keeping tracks of the variables already used. In the case where variable names can be reduced to natural numbers, this can be easily done with a reference that is incremented each time a fresh variable is needed. The translation is thus annotated by a substitution σ which binds names from the source language with names in the target language. For instance, the translation of a typing context $a : A, \alpha : A^{\perp\!\!\perp}, b : B$ is now:

$$[\![a : A, \alpha : A^{\perp\!\!\perp}, b : B]\!]^\sigma \quad = \quad \sigma(a) : \varepsilon \triangleright_t A,\ \sigma(\alpha) : [\![a : A]\!]^\sigma \triangleright_E A,\ \sigma(b) : [\![a : A, \alpha : A^{\perp\!\!\perp}]\!]^\sigma \triangleright_t B$$

6.3.2 The target language: System $\mathbf{F}_\Upsilon$

The target language is thus the usual λ-calculus, which is extended with stores (defined lists of pairs of a name and a term) and second-order quantification over store types. We refer to this language as System F_Υ. We assume that types contain at least a constant for each atomic type X of the original system, and we still denote this constant by X. This allows us to define an embedding ι from the original type system to this one by:

$$\iota(X) = X \qquad\qquad \iota(A \to B) = \iota(A) \to \iota(B).$$

The syntax for terms and types is given by:

$$\begin{array}{rcl|rcl}
t, u & ::= & \boldsymbol{k} \mid x \mid \lambda x.t \mid tu \mid \tau & A, B & ::= & X \mid \bot \mid \Upsilon \triangleright_\tau \Upsilon' \mid A \to B \mid \forall Y \diamond \Upsilon.\, A \\
& & \mid \mathtt{let}\, x_{\tau_0}, x, x_{\tau_1} = \mathtt{split}\ \tau''\, y\, \mathtt{in}\, t & \Upsilon, \Upsilon' & ::= & \varepsilon \mid (x : A) \mid (x : A^{\perp\!\!\perp}) \mid Y \mid \Upsilon, \Upsilon' \\
\tau, \tau' & ::= & \varepsilon \mid \tau[x := t] & \Gamma, \Gamma' & ::= & \varepsilon \mid \Gamma, x : A \mid \Gamma, Y <: \Upsilon
\end{array}$$

We introduce a new symbol $\Upsilon \triangleright_\tau \Upsilon'$ to denote the fact that a store has a type conditioned by Υ (which should be the type of the head of the list). In order to ease the notations, we will denote Υ instead of $\varepsilon \triangleright_\tau \Upsilon$ in the sequel. On the contrary, $\Upsilon \triangleright_t A$ is a shorthand (defined in Figure 6.6). The type system is given in Figure 6.5 where we assume that a name can only occur once both in typing contexts Γ and stores types Υ.

Remark 6.27. We shall make a few remarks about our choice of rules for typing stores. First, observe that we force elements of the store to have types of the form $\Upsilon \triangleright_t A$, that is having the structure of types obtained through the CPS translation. Even though this could appear as a strong requirement, it appears naturally when giving a computational contents to the inclusion $\Upsilon <: '\Upsilon$ with De Bruijn levels (see Section 6.4.4). Indeed, a De Bruijn level (just as a name) can be understood as a pointer to a

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

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

$$\dfrac{\Gamma, x_{\tau_0} : \Upsilon_0, x : \Upsilon_0 \triangleright_t A, x_{\tau_1} : (\Upsilon_0, y : A) \triangleright_\tau \Upsilon_1 \vdash t : B \quad \Gamma \vdash \tau : \Upsilon_0, y : A, \Upsilon_1}{\Gamma; \Sigma \vdash \mathtt{let}\, x_{\tau_0}, x, x_{\tau_1} = \mathtt{split}\ \ \mathtt{as}\,(\tau)\ \mathtt{in}\ \ y\, \mathtt{in}\, t : B}\;(\mathtt{split})$$

$$\dfrac{}{\Gamma \vdash \varepsilon : \varepsilon \triangleright_\tau \varepsilon}\;(\varepsilon) \qquad \dfrac{\Gamma \vdash t : \Upsilon_0 \triangleright_t A}{\Gamma \vdash [x := t] : \Upsilon_0 \triangleright_\tau x : A}\;(\tau_t) \qquad \dfrac{\Gamma \vdash t : \Upsilon_0 \triangleright_E A}{\Gamma \vdash [x := t] : \Upsilon_0 \triangleright_\tau x : A^{\perp\!\!\perp}}\;(\tau_E)$$

$$\dfrac{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon \qquad \Gamma \vdash \tau' : (\Upsilon_0, \Upsilon) \triangleright_\tau \Upsilon'}{\Gamma \vdash \tau\tau' : \Upsilon_0 \triangleright_\tau \Upsilon, \Upsilon'}\;(\tau\tau') \qquad \dfrac{(\Upsilon' <: \Upsilon) \in \Gamma}{\Gamma \vdash \Upsilon' <: \Upsilon}\;(<:_{\mathrm{ax}}) \qquad \dfrac{}{\Gamma \vdash Y <: Y}\;(<:_Y)$$

$$\dfrac{}{\Gamma \vdash \Upsilon <: \varepsilon}\;(<:_\varepsilon) \qquad \dfrac{\Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash (\Upsilon', x : A) <: (\Upsilon, x : A)}\;(<:_1) \qquad \dfrac{\Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash \Upsilon', \Upsilon'' <: \Upsilon}\;(<:_2)$$

$$\dfrac{\Gamma \vdash \Upsilon'' <: \Upsilon' \quad \Gamma \vdash \Upsilon' <: \Upsilon}{\Gamma \vdash \Upsilon'' <: \Upsilon}\;(<:_3) \qquad \dfrac{\Gamma \vdash \tau : \Upsilon_0' \triangleright_\tau \Upsilon' \quad \Gamma \vdash \Upsilon' <: \Upsilon \quad \Gamma \vdash \Upsilon_0 <: \Upsilon_0'}{\Gamma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon}\;(\tau_{<:})$$

$$\dfrac{\Gamma[(Y_0, x : A, Y_1)/Y] \vdash t : B[(Y_0, x : A, Y_1)/Y] \quad \Gamma \vdash Y <: (\Upsilon_0, x : A, \Upsilon_1)}{\Gamma \vdash t : B}\;(<:_{\mathtt{split}})$$

Figure 6.5: Typing rules of System F_Υ

particular cell of the store. Therefore, we need to update pointers when inserting a new element (as in Proposition 6.32). Such an operation would not have any sense (and in particular would be ill-typed) for an element that is not of type $\Upsilon \triangleright_t A$. One could circumvent this by tagging each cell of the store with a flag (using a sum type) indicating whether the corresponding elements have a type of this form or not. Second, note that each element of the store has a type depending on the type of the head of the store. Once again, this is natural and only reflects what was already happening in the source language or within the realizability interpretation. ⌟

The translation of judgments and types is given in Figure 6.6, where we made explicit the renaming procedure from the $\overline{\lambda}_{[lv\tau\star]}$-calculus to the target language. We denote by $\sigma \fatsemi \Gamma$ the fact that σ is a substitution suitable to rename every names present in Γ.

As for the reduction rules of the language, there is only two of them, namely the usual β-reduction and the split of a store with respect to a name:

$$\begin{array}{rcll} \lambda x.t\,u & \to & t[u/x] & \\ \mathtt{let}\, x_0, x, x_1 = \mathtt{split}\ \tau\, y\, \mathtt{in}\, t & \to & t[\tau_0/x_0, u/x, \tau_1/x_1] & \qquad (\text{where } \tau = \tau_0[y := u]\tau_1) \end{array}$$

6.3.3 The typed translation

We consider in this section that we dispose of a generator of fresh names (for instance a global counter) and use names explicitly both in the language (for stores) and in the type system (for their types). The

$$\begin{array}{lclcll}
[\![\Gamma \vdash_e e : A^{\perp\!\!\!\perp}]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![e]\!]_e^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_e \iota(A)) & \\
[\![\Gamma \vdash_t t : A]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![t]\!]_t^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_t \iota(A)) & \\
[\![\Gamma \vdash_E E : A^{\perp\!\!\!\perp}]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![E]\!]_E^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_E \iota(A)) & \\
[\![\Gamma \vdash_V V : A]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![V]\!]_V^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_V \iota(A)) & \\
[\![\Gamma \vdash_F F : A^{\perp\!\!\!\perp}]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![F]\!]_F^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_F \iota(A)) & \\
[\![\Gamma \vdash_v v : A]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![v]\!]_v^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_v \iota(A)) & \\
[\![\Gamma \vdash_c c]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![c]\!]_c^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_c \bot) & \\
[\![\Gamma \vdash_l l]\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ [\![l]\!]_l^\sigma : [\![\Gamma]\!]_\Gamma^\sigma \triangleright_c \bot) & \\
[\![\Gamma \vdash_\tau \tau : \Gamma']\!] & \triangleq & \forall \sigma,\ \sigma \mathrel{\mathfrak{s}} \Gamma & \Rightarrow & (\vdash\ \tau' : [\![\Gamma]\!]_\Gamma^{\sigma'} \triangleright_\tau [\![\Gamma']\!]_\Gamma^{\sigma'}) & (\text{where } \tau',\sigma' = [\![\tau]\!]_\tau^\sigma)
\end{array}$$

$$\sigma \mathrel{\mathfrak{s}} \Gamma \triangleq \sigma \text{ injective } \wedge \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\sigma)$$

$$[\![\Gamma, a : A]\!]_\Gamma^\sigma \triangleq [\![\Gamma]\!]_\Gamma^\sigma, \sigma(a) : \iota(A) \qquad [\![\Gamma, \alpha : A^{\perp\!\!\!\perp}]\!]_\Gamma^\sigma \triangleq [\![\Gamma]\!]_\Gamma^\sigma, \sigma(\alpha) : \iota(A)^{\perp\!\!\!\perp} \qquad [\![\varepsilon]\!]_\Gamma^\sigma \triangleq \varepsilon$$

$$\begin{array}{ll|ll}
\Upsilon \triangleright_c A & \triangleq \forall Y <: \Upsilon.\ Y \to \bot & \Upsilon \triangleright_V A & \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_F A) \to \bot \\
\Upsilon \triangleright_e A & \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_t A) \to \bot & \Upsilon \triangleright_F A & \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_v A) \to \bot \\
\Upsilon \triangleright_t A & \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_E A) \to \bot & \Upsilon \triangleright_v A \to B & \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_t A) \to (Y \triangleright_E B) \to \bot \\
\Upsilon \triangleright_E A & \triangleq \forall Y <: \Upsilon.\ Y \to (Y \triangleright_V A) \to \bot & \Upsilon \triangleright_v X & \triangleq X
\end{array}$$

Figure 6.6: Translation of judgments and types

next section will be devoted to the presentation of the translation using De Bruijn levels instead of names.

The translation of terms is given in Figure 6.7 where we assume that for each constant $\boldsymbol{k}$ of type X (resp. co-constant $\boldsymbol{\kappa}$ of type $A^{\perp\!\!\!\perp}$) of the source system, we have a constant of type X in the signature $\mathcal{S}$ of target language, constant that we also denote by $\boldsymbol{k}$ (resp. $\boldsymbol{\kappa}$ of type $A \to \bot$). Except for the explicit renaming, the translation is the very same as in Ariola *et al.*, hence their results are preserved with our translation. In particular, if two closures l, l' are such that $l \to l'$, then[12] $[\![l]\!]_l^\sigma =_{\beta,\eta} [\![l']\!]_l^\sigma$ (see [4, Theorem 6]).

We first prove a few technical results that we will use afterwards in the proof of the main theorem.

Lemma 6.28 (Suitable substitution)**.** *For all σ and Γ such that σ is suitable for Γ, if τ is a store such that $\Gamma \vdash_\tau \tau : \Gamma'$ for some Γ', if $\tau', \sigma' = [\![\tau]\!]_\tau^\sigma$ then σ' is suitable for Γ, Γ' and $[\![\Gamma]\!]_\Gamma^\sigma = [\![\Gamma]\!]_\Gamma^{\sigma'}$.*

Proof. Obvious from the definition. □

Lemma 6.29 (Subtyping identity)**.** *The following rule is admissible:* $\dfrac{}{\Sigma \vdash \Upsilon <: \Upsilon}$

Proof. Straightforward induction on the structure of Υ, applying repeatedly the $(<:_1)$-rule (or the $(<:_Y)$-rule). □

[12] Such a statement could be refined to prove that that the translation preserves the reduction. As in the call-by-name and call-by-value cases (see Proposition 4.18), it would require to define a translation at each level ($e, t, \ldots$) for commands, to finally prove that if $c_\iota\tau \xrightarrow{1} c_o\tau'$, then $[\![c\tau]\!]_\iota^\sigma \xrightarrow{+} [\![c\tau']\!]_o^{\sigma'}$. We claim that this would not present any specific difficulty, but that it is no longer worth bothering ourself with such a proof since we already proved the normalization.

$$
\begin{array}{lcll}
\llbracket \boldsymbol{k} \rrbracket^{\sigma}_{v} & \triangleq & \boldsymbol{k} & \\
\llbracket \lambda x.t \rrbracket^{\sigma}_{v}\ \tau\, u\, E & \triangleq & \llbracket t \rrbracket^{\sigma[x:=n]}_{t}\ \ \tau[n := u]\ E & (n \text{ fresh}) \\
\\
\llbracket \boldsymbol{\kappa} \rrbracket^{\sigma}_{F} & \triangleq & \boldsymbol{\kappa} & \\
\llbracket t \cdot E \rrbracket^{\sigma}_{F}\ \tau\, v & \triangleq & v\ \tau\ \llbracket t \rrbracket^{\sigma}_{t}\ \llbracket E \rrbracket^{\sigma}_{E} & \\
\\
\llbracket v \rrbracket^{\sigma}_{V}\ \tau\ F & \triangleq & F\ \tau\ \llbracket v \rrbracket^{\sigma}_{v} & \\
\llbracket x \rrbracket^{\sigma}_{V}\ \tau[\sigma(x) := t]\tau'\ F & \triangleq & t\ \tau\ (\lambda\tau\lambda V.V\ \tau[\sigma(x) :=\uparrow^{t} V]\tau'\ F) & (\text{with } \uparrow^{t} V = \lambda\tau E.E\,\tau\,V) \\
\\
\llbracket \alpha \rrbracket^{\sigma}_{E}\ \tau[\sigma(\alpha) := E]\tau'\ V & \triangleq & E\ \tau[\sigma(\alpha) := E]\tau'\ V & \\
\llbracket \tilde{\mu}[x].\langle x \| F\rangle\tau' \rrbracket^{\sigma}_{E}\ \tau\ V & \triangleq & V\ \tau[n :=\uparrow^{t} V]\tau''\ \llbracket F \rrbracket^{\sigma'}_{F} & (\text{where } n \text{ fresh}, \tau'', \sigma' = \llbracket \tau \rrbracket^{\sigma[x:=n]}_{\tau}) \\
\\
\llbracket V \rrbracket^{\sigma}_{t}\ \tau\ E & \triangleq & E\ \tau\ \llbracket V \rrbracket^{\sigma}_{V} & \\
\llbracket \mu\alpha.c \rrbracket^{\sigma}_{t}\ \tau\ E & \triangleq & \llbracket c \rrbracket^{\sigma[\alpha:=n]}_{c}\ \tau[n := E] & (n \text{ fresh}) \\
\\
\llbracket E \rrbracket^{\sigma}_{e}\ \tau\ t & \triangleq & t\ \tau\ \llbracket E \rrbracket^{\sigma}_{E} & \\
\llbracket \tilde{\mu}x.c \rrbracket^{\sigma}_{e}\ \tau\ t & \triangleq & \llbracket c \rrbracket^{\sigma[x:=n]}_{c}\ \tau[n := t] & (n \text{ fresh}) \\
\\
\llbracket \langle t \| e \rangle \rrbracket^{\sigma}_{c}\ \tau & \triangleq & \llbracket e \rrbracket^{\sigma}_{e}\ \tau\ \llbracket t \rrbracket^{\sigma}_{t} & \\
\llbracket c\ \tau \rrbracket^{\sigma}_{l}\ \tau_0 & \triangleq & \llbracket c \rrbracket^{\sigma'}_{c}\ \tau_0\tau' & (\text{where } \tau', \sigma' = \llbracket \tau \rrbracket^{\sigma}_{\tau}) \\
\\
\llbracket \varepsilon \rrbracket^{\sigma}_{\tau} & \triangleq & \varepsilon, \sigma & \\
\llbracket \tau'[x := t] \rrbracket^{\sigma}_{\tau} & \triangleq & \tau'[n := \llbracket t \rrbracket^{\sigma'}_{t}], \sigma[x := n] & (\text{where } \tau', \sigma' = \llbracket \tau \rrbracket^{\sigma}_{\tau},\ n \text{ fresh}) \\
\llbracket \tau'[\alpha := E] \rrbracket^{\sigma}_{\tau} & \triangleq & \tau'[n := \llbracket E \rrbracket^{\sigma'}_{E}], \sigma[\alpha := n] & (\text{where } \tau', \sigma' = \llbracket \tau \rrbracket^{\sigma}_{\tau},\ n \text{ fresh})
\end{array}
$$

Figure 6.7: Translation of terms

Lemma 6.30 (Weakening). *The following rule is admissible:*

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

Proof. Straightforward induction on typing derivations. □

Lemma 6.31 (Terms subtyping). *The following rule is admissible:*

$$\frac{\Gamma \vdash t : \forall Y <: \Upsilon_0.A \quad \Gamma \vdash \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash t : \forall Y <: \Upsilon_1.A}\ <:_{\forall}$$

Proof. We can derive:

$$\dfrac{\dfrac{\Gamma, X <: \Upsilon_1 \vdash t : \forall Y <: \Upsilon_0.A \qquad \dfrac{\dfrac{}{\Gamma, Y <: \Upsilon_1 \vdash Y <: \Upsilon_1}\ (<:_{\mathsf{ax}}) \qquad \Gamma \vdash \Upsilon_1 <: \Upsilon_0}{\Gamma, Y <: \Upsilon_1 \vdash Y <: \Upsilon_0}\ (<:_3)}{\Gamma, Y <: \Upsilon_1 \vdash t : A}\ (\forall_E) \qquad Y \notin FV(\Gamma)}{\Gamma \vdash t : \forall Y <: \Upsilon_1.A}\ (\forall_I)$$

where we use Lemma 6.30 to weaken $\Gamma, X <: \Upsilon_1$ to Γ.

□

Corollary 6.32. *For any level o of the hierarchy e, t, E, V, F, v, the following rule is admissible:*

$$\frac{\Gamma \vdash t : \Upsilon_0 \triangleright_o A \quad \Gamma \vdash \Upsilon_1 <: \Upsilon_0}{\Gamma \vdash t : \Upsilon_1 \triangleright_o A}\ <:_{\triangleright}$$

Theorem 6.33 (Preservation of typing)**.** *The translation is well-typed, i.e.*

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

Proof. By induction over the typing rules. Let Γ be a typing context and σ be a suitable translation of names of Γ. We (ab)use of Lemma 6.30 to make the derivations more compact by systematically weakening contexts as soon as possible. We also compact the first $\forall$- and λ-introductions in one rule.

1. Strong values

- **Case** $[\![k]\!]_v^\sigma$. $[\![k]\!]_v^\sigma = k$, which has the desired type by hypothesis.

- **Case** $[\![\lambda x_i.t]\!]_v^\sigma$. In the source language, we have:

$$\frac{\Gamma, x : A \vdash_t t : B}{\Gamma \vdash_v \lambda x : A \to B}$$

Hence, if n is fresh (w.r.t. σ), $\sigma[x := n]$ is suitable for $\Gamma, x : A$, and we get by induction a proof Π_t of $[\![t]\!]_t^{\sigma[x:=n]} : [\![\Gamma, x : A]\!]_\Gamma^{\sigma[x:=n]} \triangleright_t \iota(B)$. Observing that $[\![\Gamma, x : A]\!]_\Gamma^{\sigma[x:=n]} = [\![\Gamma]\!]_\Gamma^\sigma, n : \iota(A)$ we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\vdash [\![t]\!]_t^{\sigma[x:=n]} : [\![\Gamma, x : A]\!]_\Gamma^{[\sigma,x:=n]} \triangleright_t \iota(B)} \quad \dfrac{\dfrac{}{Y <: [\![\Gamma]\!]_\Gamma^\sigma \vdash Y <: [\![\Gamma]\!]_\Gamma^\sigma}\,(<:_{\mathsf{ax}})}{Y <: [\![\Gamma]\!]_\Gamma^\sigma \vdash Y, n : A <: [\![\Gamma]\!]_\Gamma^\sigma, n : A}\,(<:_2)}{Y <: [\![\Gamma]\!]_\Gamma^\sigma \vdash [\![t]\!]_t^{\sigma[x:=n]} : Y, n : A \to Y, n : A \triangleright_E \iota(B) \to \bot}\,(\forall_E) \quad \Pi_\tau}{Y <: [\![\Gamma]\!]_\Gamma^\sigma, \tau : Y, u : Y \triangleright_t \iota(A); \vdash [\![t]\!]_t\ \tau[u] : Y, n : A \triangleright_E \iota(B) \to \bot}\,(@) \quad \Pi_E}{Y <: [\![\Gamma]\!]_\Gamma^\sigma, \tau : Y, u : Y \triangleright_t \iota(A), E : Y \triangleright_E \iota(B) \vdash [\![t]\!]_t^{\sigma[x:=n]}\ \tau[u]\ E : \bot}\,(@)}{\vdash \lambda \tau u E.[\![t]\!]_t^{\sigma[x:=n]}\ \tau[u]\ E : \forall Y <: [\![\Gamma]\!]_\Gamma^\sigma . Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\,(\lambda)$$

where:

- Π_τ is the following subproof:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{u : Y \triangleright_t \iota(A) \vdash u : Y \triangleright_t \iota(A)}\,(\mathrm{Ax})}{Y \triangleright_t \iota(A) \vdash [n := u] : Y \triangleright_\tau \iota(A)}\,(\tau_t)}{\tau : Y, u : Y \triangleright_t \iota(A); \vdash \tau[n := u] : Y, n : A}\,(\tau\tau')$$

- Π_E is the following proof (derivable using Corollary 6.32):

$$\dfrac{\dfrac{}{E : Y \triangleright_E \iota(B) \vdash E : (Y, n : \iota(A)) \triangleright_E \iota(B)}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{}{\vdash Y <: Y}\,(<:_{\mathsf{ax}})}{\vdash (Y, n : \iota(A)) <: Y}\,(<:_2)}{E : Y \triangleright_E \iota(B) \vdash E : (Y, n : \iota(A)) \triangleright_E \iota(B)}\,<:_{\triangleright}$$

2. Forcing contexts

- **Case** $[\![\kappa]\!]_F^\sigma$. $[\![\kappa]\!]_F^\sigma = \kappa$, which has the desired type by hypothesis.

- **Case** $\llbracket t.E \rrbracket^\sigma_F$. In the source language, we have:

$$\dfrac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_E E : B^{\perp\!\!\perp}}{\Gamma \vdash_F t \cdot E : (A \to B)^{\perp\!\!\perp}}$$

Therefore, we obtain by induction a proof of $\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_t \iota(A)$ (and a proof of $\vdash \llbracket E \rrbracket_t : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_E \iota(B)$) that can be turned (using Corollary 6.32) into a proof Π_t of $Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash \llbracket t \rrbracket_t : Y \triangleright_t \iota(A)$ for any Y (resp. Π_E of $Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash \llbracket E \rrbracket_t : Y \triangleright_E \iota(B)$). Thus, we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v : Y \triangleright_v \iota(A) \to \iota(B)}\,(\text{Ax}) \qquad \dfrac{}{\vdash Y <: Y}\,(<:_{\text{ax}})}{v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v : Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E B \to \perp}\,(\forall_E) \qquad \dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax})}{\tau : Y, v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v\ \tau : Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \perp}\,(@) \qquad \boldsymbol{\Pi_t}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y, v : Y \triangleright_v (\iota(A) \to \iota(B)) \vdash v\ \tau\ \llbracket t \rrbracket_t : Y \triangleright_E \iota(B) \to \perp}\,(@) \qquad \boldsymbol{\Pi_E}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y, v : Y \triangleright_v (\iota(A) \to \iota(B))(\vdash v\ \tau\ \llbracket t \rrbracket_t\ \llbracket E \rrbracket^\sigma_E : \perp}\,(@)}{\vdash \lambda \tau v. v\ \tau\ \llbracket t \rrbracket_t\ \llbracket E \rrbracket^\sigma_E : \forall Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma . Y \to Y \triangleright_v (\iota(A) \to \iota(B)) \to \perp}\,(\lambda)$$

3. Weak values

- **Case** $\llbracket v \rrbracket_V$. In the source language, we have:

$$\dfrac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}$$

Then we have by induction hypothesis a proof Π_v of $\vdash \llbracket v \rrbracket^\sigma_v : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_v \iota(A)$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F : Y \triangleright_F \iota(A) \vdash F : Y \triangleright_F \iota(A)}\,(\text{Ax}) \qquad \dfrac{}{\vdash Y <: Y}\,(<:_{\text{ax}})}{F : Y \triangleright_F \iota(A) \vdash F : Y \to Y \triangleright_v \iota(A) \to \perp}\,(\forall_E) \qquad \dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax})}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y, F : Y \triangleright_F \iota(A) \vdash F\ \tau : Y \triangleright_v \iota(A) \to \perp}\,(@) \qquad \dfrac{\boldsymbol{\Pi_v} \qquad \dfrac{}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma}\,(\text{Ax})}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash \llbracket v \rrbracket^\sigma_v : Y \triangleright_v \iota(A)}\,<:_{\triangleright}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y, F : Y \triangleright_F \iota(A) \vdash F\ \tau\ \llbracket v \rrbracket^\sigma_v : \perp}\,(@)}{\vdash \lambda \tau F. F\ \tau\ \llbracket v \rrbracket^\sigma_v : \forall Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma . Y \to Y \triangleright_F \iota(A) \to \perp}\,(\lambda)$$

where we used Corollary 6.32 on the right part of the proof. Observe that $\uparrow^t V$ is in fact independent of the level t and that we could as well have written $\llbracket v \rrbracket_V = \uparrow \llbracket v \rrbracket^\sigma_v$. We thus proved the admissibility of the following rule:

$$\dfrac{\Gamma \vdash V : \Upsilon \triangleright_V A}{\Gamma \vdash \uparrow^t V : \Upsilon \triangleright_t A}\,(\uparrow)$$

- **Case** $\llbracket x \rrbracket_V$. In the source language, we have:

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

so that Γ is of the form $\Gamma_0, x : A, \Gamma_1$. By definition, we have:

$$\llbracket x \rrbracket_V = \lambda \tau F. \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \tau_0\ (\lambda \tau_0' V. V\ \tau_0'[n := \uparrow^t V] \tau_1\ F) \qquad \text{where } n = \sigma(x)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t : Y_0 \triangleright_t \iota(A) \vdash t : Y_0 \triangleright_t \iota(A)}\,(\text{Ax}) \qquad \dfrac{}{\vdash Y_0 <: Y_0}\,(<:_{\text{ax}})}{t : Y_0 \triangleright_t \iota(A) \vdash t : Y_0 \to Y_0 \triangleright_E \iota(A) \to \perp}\,(\forall_E) \qquad \dfrac{}{\tau_0 : Y_0 \vdash \tau_0 : Y_0}\,(\text{Ax})}{\tau_0 : Y_0, t : Y_0 \triangleright_t A \vdash t\ \tau_0 : Y_0 \triangleright_E \iota(A) \to \perp}\,(@) \qquad \boldsymbol{\Pi_E}}{\tau_0 : Y_0, t : Y_0 \triangleright_t \iota(A), \tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash t\ \tau_0\ E : \perp}\,(@)}{\tau : (Y_0, n : \iota(A), Y_1), F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \tau_0\ E : \perp}\,(\mathtt{split}) \qquad \boldsymbol{\Pi_Y}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y, F : Y \triangleright_F \iota(A) \vdash \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \tau_0\ E : \perp}\,(<:_{\mathtt{split}})}{\vdash \lambda \tau F. \mathtt{let}\, \tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \tau_0\ E : \forall Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma . Y \to Y \triangleright_F \iota(A) \to \perp}\,(\lambda)$$

where:

- Π_Y is simply the axiom rule:

$$\dfrac{}{Y <: (\llbracket\Gamma_0\rrbracket^\sigma_\Gamma, n : \iota(A), \llbracket\Gamma_1\rrbracket^\sigma_\Gamma) \vdash Y <: (\llbracket\Gamma_0\rrbracket^\sigma_\Gamma, n : \iota(A), \llbracket\Gamma_1\rrbracket^\sigma_\Gamma)}\,(<:_{\text{ax}})$$

- $E = (\lambda\tau_0'V.V\ \tau_0'[n := V]\tau_1\ F)$ and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_V \iota(A) \vdash\uparrow^t V : Y_0' \triangleright_t \iota(A)}\,(\text{Ax}) \quad \dfrac{}{\vdash Y_0', n : A, Y_1 <: Y_0', n : A}\,(\text{Ax})}{V : Y_0' \triangleright_V \iota(A) \vdash V : (Y_0', n : A, Y_1) \to (Y_0', n : A, Y_1) \triangleright_E \iota(A) \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi_\tau}}{\tau_1 : (Y_0, n : A) \triangleright_\tau Y_1, Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash V\ \tau_0'[n :=\uparrow^t V]\tau_1 : (Y_0', n : \iota(A), Y_1) \triangleright_F \iota(A) \to \bot}\,(@) \quad \boldsymbol{\Pi_F}}{\tau_1 : (Y_0, n : A) \triangleright_\tau Y_1, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A), Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash V\ \tau_0'[n :=\uparrow^t V]\tau_1\ F : \bot}\,(@)}{\tau_1 : (Y_0, n : A) \triangleright_\tau Y_1, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash \lambda\tau_0'V.V\ \tau_0'[n :=\uparrow^t V]\tau_1\ F : Y_0 \triangleright_E \iota(A)}\,(\lambda)$$

- Π_F is the following proof, obtained by Corollary 6.32:

$$\dfrac{\dfrac{}{F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A)}\,(\text{Ax}) \quad \dfrac{\begin{array}{c}\dfrac{\dfrac{}{Y_0' <: Y_0 \vdash Y_0' <: Y_0}\,(<:_{\text{ax}})}{}\,(<:_1) \\ \vdots\end{array}}{Y_0' <: Y_0 \vdash (Y_0', n : \iota(A), Y_1) <: (Y_0, n : \iota(A), Y_1)}\,(<:_1)}{Y_0' <: Y_0, F : (Y_0, n : \iota(A), Y_1) \triangleright_F \iota(A) \vdash F : (Y_0', n : \iota(A), Y_1) \triangleright_F \iota(A)}\,{<:_\triangleright}$$

- Π_τ is the following derivation

$$\dfrac{\dfrac{\dfrac{}{\tau_0' : Y_0' \vdash \tau_0' : Y_0'}\,(\text{Ax}) \quad \dfrac{\dfrac{\dfrac{}{V : Y_0' \triangleright_V \iota(A) \vdash V : Y_0' \triangleright_V A}\,(\text{Ax})}{V : Y_0' \triangleright_V \iota(A) \vdash\uparrow^t V : Y_0' \triangleright_t A}\,(\uparrow)}{V : Y_0' \triangleright_V \iota(A) \vdash [n :=\uparrow^t V] : Y_0' \triangleright_\tau n : \iota(A)}\,(\tau_t)}{Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[n := V] : Y_0', n : \iota(A)}\,(\tau\tau') \quad \boldsymbol{\Pi_{\tau_1}}}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1, Y_0' <: Y_0, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[n := V]\tau_1 : Y_0', n : A, Y_1}\,(\tau_{<:})$$

- Π_{τ_1} is the following derivation:

$$\dfrac{\dfrac{}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1 \vdash \tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1}\,(\text{Ax}) \quad \dfrac{\dfrac{}{Y_0' <: Y_0 \vdash Y_0' <: Y_0}\,(<:_{\text{ax}})}{Y_0' <: Y_0 \vdash Y_0', n : \iota(A) <: Y_0, n : \iota(A)}\,(<:_1)}{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1, Y_0' <: Y_0 \vdash \tau_1 : (Y_0', n : \iota(A)) \triangleright_\tau Y_1}\,(\tau_{<:})$$

4. Catchable contexts

- **Case** $\llbracket F\rrbracket^\sigma_E$. This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case** $\llbracket\tilde\mu[x].\langle x\|F\rangle\tau'\rrbracket^\sigma_E$. In the source language, we have:

$$\dfrac{\Gamma, x : A, \Gamma' \vdash_F F : A^{⫫} \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_E \tilde\mu[x].\langle x\|F\rangle\tau : A^{⫫}}$$

If n is fresh (w.r.t σ), $\sigma[x := n]$ is suitable for $\Gamma, x : A$, and we then have by induction hypothesis a proof of $\vdash \tau'' : \llbracket\Gamma, x : A\rrbracket^{\sigma'} \triangleright_\tau \llbracket\Gamma'\rrbracket^{\sigma'}$ and a proof Π_F of $\vdash \llbracket F\rrbracket^{\sigma'}_F : \llbracket\Gamma, x : A\rrbracket^{\sigma'} \triangleright_F \iota(A)$ where $\tau'', \sigma' = \llbracket\tau'\rrbracket^{\sigma,[x:=n]}$ for some fresh n. We can thus derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{V : Y \triangleright_V \iota(A) \vdash Y \triangleright_V \iota(A)}\,(\text{Ax}) \quad \dfrac{\dfrac{}{\vdash Y <: Y}\,(<:_{\text{ax}})}{\vdash Y, n : \iota(A), \llbracket\Gamma'\rrbracket^{\sigma'}_\Gamma <: Y}\,(<:_2)}{V : Y \triangleright_V \iota(A) \vdash V : (Y, n : \iota(A), \llbracket\Gamma'\rrbracket^{\sigma'}_\Gamma) \to (Y, n : \iota(A), \llbracket\Gamma'\rrbracket^{\sigma'}_\Gamma) \triangleright_F \iota(A) \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi_\tau}}{\tau : Y, V : Y \triangleright_V \iota(A) \vdash V\ \tau[n :=\uparrow^t V]\tau'' : (Y, n : \iota(A), \llbracket\Gamma'\rrbracket^{\sigma'}_\Gamma \iota(A) \triangleright_F \to \bot}\,(@) \quad \boldsymbol{\Pi_F}}{Y <: \llbracket\Gamma\rrbracket^{\sigma'}_\Gamma, \tau : Y, V : Y \triangleright_V \iota(A) \vdash V\ \tau[n :=\uparrow^t V]\tau''\ \llbracket F\rrbracket^{\sigma'}_F : \bot}\,(@)}{\vdash \lambda\tau V.V\ \tau[n :=\uparrow^t V]\tau''\ \llbracket F\rrbracket^{\sigma'}_F : \forall Y <: \llbracket\Gamma\rrbracket^{\sigma'}_\Gamma.Y \to Y \triangleright_V \iota(A) \to \bot}\,(\lambda)$$

where:

- Π_F is the following proof, derived using Corollary 6.32 and Lemma 6.28:

$$\dfrac{\vdash \llbracket F\rrbracket_F^{\sigma'} : \llbracket \Gamma, n:\iota(A), \Gamma'\rrbracket_\Gamma^{\sigma'} \rhd_F \iota(A) \qquad \dfrac{\dfrac{}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'} \vdash Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'}}\,(\text{Ax})}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'} \vdash Y, n:\iota(A), \llbracket\Gamma'\rrbracket_\Gamma^{\sigma'} <: \llbracket\Gamma, n:\iota(A),\Gamma'\rrbracket_\Gamma^{\sigma'}}\,(<:_1)}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'} \vdash \llbracket F\rrbracket_F^{\sigma'} : Y, n:\iota(A), \llbracket\Gamma'\rrbracket_\Gamma^{\sigma'} \rhd_F \iota(A)}\,(\forall_E)$$

- Π_τ is the following proof:

$$\dfrac{\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{V : Y \rhd_V \iota(A) \vdash V : Y \rhd_V \iota(A)}\,(\text{Ax})}{V : Y \rhd_V \iota(A) \vdash {\uparrow^t} V : Y \rhd_t \iota(A)}\,(\uparrow)}{V : Y \rhd_V \iota(A) \vdash [n := V] : Y \rhd_\tau n : \iota(A)}\,(\tau_t)}{\tau : Y, V : Y \rhd_V \iota(A) \vdash \tau[n :=\uparrow^t V] : Y, n : \iota(A)}\,(\tau\tau') \qquad \boldsymbol{\Pi_{\tau'}}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma'}, \tau : Y, V : Y' \rhd_V \iota(A) \vdash \tau[n :=\uparrow^t V]\llbracket\tau'\rrbracket_\tau^{\sigma[x:=n]} : (Y, n : \iota(A), \llbracket\Gamma'\rrbracket^{\sigma[x:=n]})}\,(\tau\tau')$$

- $\Pi_{\tau'}$ is the following proof, obtained from the induction hypothesis for τ':

$$\dfrac{\vdash \llbracket\tau'\rrbracket_\tau^{\sigma[x:=n]} : \llbracket\Gamma\rrbracket_\Gamma^{\sigma}, n : \iota(A) \rhd_\tau \llbracket\Gamma'\rrbracket^{\sigma[x:=n]} \qquad \dfrac{\overline{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma} \vdash Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma}}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma} \vdash Y, n : \iota(A) <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma}, n : \iota(A)}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma} \vdash \llbracket\tau'\rrbracket_\tau^{\sigma[x:=n]} : Y, n : \iota(A) \rhd_\tau \llbracket\Gamma'\rrbracket^{\sigma[x:=n]}}$$

5. Terms

- **Case** $\llbracket V\rrbracket_t$. This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case** $\llbracket \mu\alpha.c\rrbracket_t$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\dfrac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash_c c}{\Gamma \vdash_t \mu\alpha.c : A}$$

If n is fresh (w.r.t σ), $\sigma[\alpha := n]$ is suitable for $\Gamma, \alpha : A^{\perp\!\!\!\perp}$, and we then have by induction hypothesis a proof Π_c of $\vdash \llbracket c\rrbracket_c^{[\sigma,x:=n]} : \llbracket\Gamma, \alpha : A^{\perp\!\!\!\perp}\rrbracket_\Gamma^{\sigma[\alpha:=n]} \rhd_c \perp$. We can thus derive, using Lemma 6.28 to identify $\llbracket\Gamma\rrbracket_\Gamma^{\sigma}$ and $\llbracket\Gamma\rrbracket_\Gamma^{\sigma,[\alpha:=n]}$:

$$\dfrac{\dfrac{\dfrac{\vdash \llbracket c\rrbracket_c^{\sigma[\alpha:=n]} : \llbracket\Gamma, \alpha : A^{\perp\!\!\!\perp}\rrbracket_\Gamma^{\sigma[\alpha:=n]} \rhd_c \perp \qquad \dfrac{\dfrac{}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma} \vdash Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma[\alpha:=n]}}\,(<:_{\text{ax}})}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma} \vdash (Y, n : \iota(A)^{\perp\!\!\!\perp}) <: \llbracket\Gamma, \alpha : A^{\perp\!\!\!\perp}\rrbracket^{\sigma[\alpha:=n]}}\,(<:_1)}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma} \vdash \llbracket c\rrbracket_c^{\sigma[\alpha:=n]} : (Y, n : \iota(A)^{\perp\!\!\!\perp}) \to \perp}\,(\forall_E) \qquad \boldsymbol{\Pi_\tau}}{Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma}, \tau : Y, E : Y \rhd_E \iota(A) \vdash \llbracket c\rrbracket_c^{\sigma[\alpha:=n]}\ \tau[n := E] : \perp}\,(@)}{\vdash \lambda\tau E.\llbracket c\rrbracket_c^{\sigma[\alpha:=n]}\ \tau[n := E] : \forall Y <: \llbracket\Gamma\rrbracket_\Gamma^{\sigma}.Y \to Y \rhd_E \iota(A) \to \perp}\,(\lambda)$$

where Π_τ is the following derivation:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\text{Ax}) \qquad \dfrac{\dfrac{}{E : Y \rhd_E \iota(A) \vdash E : Y \rhd_E \iota(A)}\,(\text{Ax})}{E : Y \rhd_E \iota(A) \vdash [n := E] : (Y \rhd_\tau n : \iota(A)^{\perp\!\!\!\perp})}\,(\tau_t)}{\tau : Y, E : Y \rhd_E \iota(A) \vdash \tau[n := E] : (Y, n : \iota(A)^{\perp\!\!\!\perp})}\,(\tau\tau')$$

6. Contexts

- **Case** $\llbracket E \rrbracket_e$. This case is similar to the case $\llbracket v \rrbracket_V$.

- **Case** $\llbracket \tilde{\mu}x.c \rrbracket_e$. This case is similar to the case $\llbracket \mu\alpha.c \rrbracket_t$.

7. Commands

- **Case** $\llbracket \langle t \| e \rangle \rrbracket_c$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

$$\frac{\Gamma \vdash_t t : A \qquad \Gamma \vdash_e e : A^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \| e \rangle}$$

We thus get by induction two proofs $\vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_e \iota(A)$ and $\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_t \iota(A)$ We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_e \iota(A) \qquad \boldsymbol{\Pi_Y}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash \llbracket e \rrbracket_e : Y \to Y \triangleright_t \iota(A) \to \bot}\,{\scriptstyle(\forall_E)} \quad \dfrac{}{\tau : Y \vdash \tau : Y}\,{\scriptstyle(\mathrm{Ax})}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y \vdash \llbracket e \rrbracket_e\ \tau : Y \triangleright_t \iota(A) \to \bot}\,{\scriptstyle(@)} \quad \dfrac{\vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket^\sigma_\Gamma \triangleright_t \iota(A) \quad \boldsymbol{\Pi_Y}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash \llbracket t \rrbracket_t : Y \triangleright_t \iota(A)}\,{\scriptstyle(\forall_E)}}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma, \tau : Y \vdash \llbracket e \rrbracket_e\ \tau\ \llbracket t \rrbracket_t : \bot}\,{\scriptstyle(@)}}{\vdash \lambda\tau.\llbracket e \rrbracket_t\ \tau\ \llbracket t \rrbracket_t : \forall Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma . Y \to \bot}\,{\scriptstyle(\lambda)}$$

where Π_Y is simply the axiom rule:

$$\frac{}{Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma \vdash Y <: \llbracket \Gamma \rrbracket^\sigma_\Gamma}\,{\scriptstyle(<:_{\mathrm{ax}})}$$

8. Closures

- **Case** $\llbracket \langle t \| e \rangle \tau \rrbracket^\sigma_l$. In the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we have:

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

We thus get by induction two proofs $\vdash \tau' : \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket^{\sigma'}_\Gamma$ and $\vdash \llbracket c \rrbracket^{\sigma'}_c : \llbracket \Gamma, \Gamma' \rrbracket^{\sigma'}_\Gamma \triangleright_c \bot$ where $\tau', \sigma' = \llbracket \tau \rrbracket^\sigma_\tau$. We can derive:

$$\dfrac{\dfrac{\dfrac{\vdash \llbracket c \rrbracket^{\sigma'}_c : \llbracket \Gamma, \Gamma' \rrbracket^{\sigma'}_\Gamma \triangleright_c \bot \quad \dfrac{\dfrac{}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \vdash Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma}\,{\scriptstyle(<:_{\mathrm{ax}})}}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \vdash Y, \llbracket \Gamma' \rrbracket^{\sigma'}_\Gamma <: \llbracket \Gamma, \Gamma' \rrbracket^{\sigma'}_\Gamma}\,{\scriptstyle(<:_1)}}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \vdash \llbracket c \rrbracket^{\sigma'}_c : Y, \llbracket \Gamma' \rrbracket^{\sigma'}_\Gamma \to \bot}\,{\scriptstyle(\forall_E)} \qquad \boldsymbol{\Pi_\tau}}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma, \tau_0 : Y \vdash \llbracket c \rrbracket^{\sigma'}_c\ \tau_0\tau' : \bot}\,{\scriptstyle(@)}}{\vdash \lambda\tau_0.\llbracket c \rrbracket^{\sigma'}_c\ \tau_0\tau' : \forall Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma . Y \to \bot}\,{\scriptstyle(\lambda)}$$

where Π_τ is the following subderivation:

$$\dfrac{\dfrac{}{\tau_0 : Y \vdash \tau_0 : Y}\,{\scriptstyle(\mathrm{Ax})} \quad \dfrac{\vdash \tau' : \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket^{\sigma'}_\Gamma \quad \dfrac{}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \vdash Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma}\,{\scriptstyle(<:_{\mathrm{ax}})}}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma \vdash \tau' : Y \triangleright_\tau \llbracket \Gamma' \rrbracket^{\sigma'}_\Gamma}\,{\scriptstyle(\tau_{<:})}}{Y <: \llbracket \Gamma \rrbracket^{\sigma'}_\Gamma, \tau_0 : Y \vdash \tau_0\tau' : Y, \llbracket \Gamma' \rrbracket^{\sigma'}_\Gamma}\,{\scriptstyle(\tau\tau')}$$

9. Stores

- **Case** $\tau[x := t]$. We only consider the case $\tau[x := t]$, the proof for the case $\tau[\alpha := E]$ is identical. This corresponds to the typing rule:

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \qquad \Gamma, \Gamma' \vdash_t t : A}{\Gamma \vdash_\tau \tau[x := t] : \Gamma', x : A}$$

By induction, we obtain two proofs of $\vdash \tau' : [\![\Gamma]\!]_\Gamma^{\sigma'} \triangleright_\tau [\![\Gamma']\!]_\Gamma^{\sigma'}$ and $\vdash [\![t]\!]_t^{\sigma'} : [\![\Gamma,\Gamma']\!]_\Gamma^{\sigma'} \triangleright_t \iota(A)$ where $\tau',\sigma' = [\![\tau]\!]_\tau^\sigma$ We can thus derive:

$$\dfrac{\vdash \tau' : [\![\Gamma]\!]_\Gamma^{\sigma'} \triangleright_\tau [\![\Gamma']\!]_\Gamma^{\sigma'} \qquad \dfrac{\vdash [\![t]\!]_t^{\sigma'} : [\![\Gamma,\Gamma']\!]_\Gamma^{\sigma'} \triangleright_t n : \iota(A)}{\vdash [n := [\![t]\!]_t^{\sigma'}] : [\![\Gamma,\Gamma']\!]_\Gamma^{\sigma'} \triangleright_\tau n : \iota(A)}\,(\tau_t)}{\vdash \tau'[n := [\![t]\!]_t^{\sigma'}] : [\![\Gamma]\!]_\Gamma^{\sigma'} \triangleright_\tau [\![\Gamma']\!]_\Gamma^{\sigma'}, n : \iota(A)}\,(\tau\tau')$$

□

Combining the preservation of reduction through the CPS and a proof of normalization of our target language (that one could obtain for instance using realizability techniques again), the former theorem would provide us with an alternative proof of normalization of the $\overline{\lambda}_{lv}$- and $\overline{\lambda}_{[lv\tau\star]}$-calculi.

6.4 Introducing De Bruijn levels

One standard way to handle issues related to α-conversion is to use De Bruijn indices [38]. In a nutshell, the De Bruijn notation is a nameless representation for λ-terms which replaces a bounded variable x by the number of λ that are crossed between the variable and its binder. For instance, the term $\lambda x.x$ is written $\lambda.0$, $\lambda xy.x$ is written $\lambda.\lambda.1$ and $\lambda x.x(\lambda y.xy)$ is written $\lambda.0(\lambda.1\,0)$. On the contrary, De Bruijn levels attributes a fixed number to λ binders (according to their "levels", that is how many former binders are crossed to reach them) and number variables in function of their binder's number. For instance, in the term $\lambda x.x(\lambda y.xy)$, the first binder λx is at top-level (level 0), while λy is at level 1. Using De Bruijn levels, this term is thus written $\lambda.1(\lambda.0\,1)$. These well-known techniques are very useful when it comes to implementation to prevent problem of α-conversion.

As we shall now see, the problem α-conversion needs to be handled carefully for the $\overline{\lambda}_{[lv\tau\star]}$-calculus and its continuation-passing-style translation, leading otherwise to non-terminating computations. This is why we needed to add explicit renaming to the translation of the previous section, since this problem was not tackled in the original translation. Another way of solving this difficulty consists in an adaptation of De Bruijn levels. Interestingly, it turns out that through the CPS, De Bruijn levels unveil some computational content related with store extensions.

6.4.1 The need for α-conversion

As for the proof of normalization, we observe in Figure 6.7 that the translation relies on names which implicitly suggests ability to perform α-conversion at run-time. Let us take a closer look at an example to better understand this phenomenon.

Example 6.34 (Lack of α-conversion). Let us consider a typed closure $\langle t \| e\rangle\tau$ such that:

$$\dfrac{\dfrac{\dfrac{\pi_t}{\Gamma \vdash_t t : A} \quad \dfrac{\pi_e}{\Gamma \vdash_e e : A^{\perp\!\!\perp}}}{\Gamma \vdash_c \langle t \| e\rangle} \quad \dfrac{\pi_\tau}{\vdash_\tau \tau : \Gamma}}{\vdash_l \langle t \| e\rangle\tau}$$

Assume that both t and e introduce a new variable x in their sub-derivations π_t and π_e, which will be the case for instance if $t = \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle$ and $e = \tilde{\mu}x.\langle x \| F\rangle$. This is compatible with previous typing derivation, however, this command would reduce (without α-conversion) as follows:

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

This command will then loop forever because of the auto-reference $[x := x]$ in the store. ⌟

This problem is reproduced through a naive CPS translation without renaming (as it was originally defined in [4]). In fact, the translation is somewhat even more problematic. Since "different" variables named x (that is variables which are bound by different binders) are translated independently (*e.g.* $[\![\langle t\|e\rangle]\!]$ is defined from $[\![e]\!]$ and $[\![t]\!]$), there is no hope to perform α-conversion on the fly during the translation. Moreover, our translation (as well as the original CPS in [4]) is defined modulo administrative translation (observe for instance that the translation of $[\![\lambda x.v]\!]_v^\sigma\ \tau\ V$ makes the λx binder vanish). Thus, the problem becomes unsolvable after the translation, as illustrated in the following example.

Example 6.35 (Lack of α-conversion in the CPS)**.** The naive translation (*i.e.* without renaming) of the same closure is again a program that will loop forever:

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

Observe that as the translation is defined modulo administrative reduction, the first equations indeed are equalities, and that when the reduction is performed, the two "different" x are not bound anymore. Thus, there is no way to achieve any kind of α-conversion to prevent the formation of the cyclic reference $[x := [\![x]\!]_V]$. ⌟

This is why we would need either to be able to perform α-conversion while executing the translation of a command, assuming that we can find a smooth way to do it, or to explicitly handle the renaming as we did in Section 8.3. As highlighted by the next example, this problem does not occur with the translation we defined, since two different fresh names are attributed to the "different" variables x.

Example 6.36 (Explicit renaming)**.** To compact the notations, we will write $[^x_m|^\alpha_\gamma|\ldots]$ for the renaming substitution $[x := m, \alpha := \gamma, \ldots]$, where we adopt the convention that the most recent binding is on written on the right. As a binding $[x := n]$ overwrites any former binding $[x := m]$, we write $[{}^{\alpha}_{\gamma}|{}^{x}_{n}]$ instead of $[{}^{x}_{m}|{}^{\alpha}_{\gamma}|{}^{x}_{n}]$.

$$\begin{aligned}
\llbracket c\varepsilon \rrbracket^{\varepsilon} &= \llbracket e \rrbracket_e^{\varepsilon}\ \varepsilon\ \llbracket t \rrbracket_t^{\varepsilon} = \llbracket \tilde{\mu}x.\langle x \| F\rangle \rrbracket_e^{\varepsilon}\ \varepsilon\ \llbracket t \rrbracket_t^{\varepsilon} \\
&= \llbracket \langle x \| F\rangle \rrbracket_c^{[{}^{x}_{m}]}\ [m := \llbracket t \rrbracket_t] \\
&= \llbracket x \rrbracket_t^{[{}^{x}_{m}]}\ [m := \llbracket t \rrbracket_t^{\varepsilon}]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]} \\
&= \llbracket \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \rrbracket_t^{[{}^{x}_{m}]}\ \varepsilon\ (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}) \\
&= \llbracket \langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}] \\
&= \llbracket \tilde{\mu}x.\langle x \| \alpha\rangle \rrbracket_e^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}]\ \llbracket u \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]} \\
&= \llbracket \langle x \| \alpha\rangle \rrbracket_c^{[x:=m,\alpha:=\gamma,x:=n]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}] \\
&= \llbracket \alpha \rrbracket_E^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}]\llbracket x \rrbracket_V^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}|{}^{x}_{n}]} \\
&= (\lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V])\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}]\ \llbracket x \rrbracket_V^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]} \\
&\to \llbracket x \rrbracket_V^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}, m := \llbracket x \rrbracket_t^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}] \\
&= \llbracket x \rrbracket_V^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}\ [\gamma := \lambda\tau\lambda V.V\ \tau[m :=\uparrow^t V]\ \llbracket F \rrbracket_F^{[{}^{x}_{m}]}, n := \llbracket u \rrbracket_t^{[{}^{x}_{m}|{}^{\alpha}_{\gamma}]}, m := \llbracket x \rrbracket_t^{[{}^{\alpha}_{\gamma}|{}^{x}_{n}]}]
\end{aligned}$$

We observe that in the end, the variable m is bound to the variable n, which is now correct. ⌟

Another way of ensuring the correctness of our translation is to correct the problem already in the $\overline{\lambda}_{[lv\tau\star]}$, using what we call De Bruijn levels. As we observed in the first example of this section, the issue arises when adding a binding $[x := ...]$ in a store that already contained a variable x. We thus need to ensure the uniqueness of names within the store. An easy way to do this consists in changing the names of variable bounded in the store by the position at which they occur in the store, which is obviously unique. Just as De Bruijn indices are pointers to the correct binder, De Bruijn levels are pointers to the correct cell of the environment. Before presenting formally the corresponding system and the adapted translation, let us take a look at the same example that we reduce using this idea. We use a mixed notation for names, writing x when a variable is bounded by a λ or a $\tilde{\mu}$, and x_i (where i is the relevant information) when it refers to a position in the store.

Example 6.37 (Reduction with De-Bruijn levels)**.** The same reduction is now safe if we replace stored variables by their De Bruijn level:

$$\begin{aligned}
\langle \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \| \tilde{\mu}x.\langle x \| F\rangle\rangle &\to \langle x_0 \| F\rangle[{}^0\mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle] \\
&\to \langle \mu\alpha.\langle u \| \tilde{\mu}x.\langle x \| \alpha\rangle\rangle \| \tilde{\mu}[x].\langle x \| F\rangle\rangle \\
&\to \langle u \| \tilde{\mu}x.\langle x \| \alpha_0\rangle\rangle[{}^0\tilde{\mu}[x].\langle x \| F\rangle] \\
&\to \langle x_1 \| \alpha_0\rangle[{}^0\tilde{\mu}[x].\langle x \| F\rangle, {}^1u] \\
&\to \langle x_1 \| \tilde{\mu}[x].\langle x \| F\rangle\rangle[\tilde{\mu}[x].\langle x \| F\rangle, {}^1u] \\
&\to \langle x_2 \| F\rangle[{}^0\tilde{\mu}[x].\langle x \| F\rangle, {}^1u, {}^2x_1] \\
&\to \langle x_1 \| \tilde{\mu}[x].\langle x \| F\rangle\rangle[{}^0\tilde{\mu}[x].\langle x \| F\rangle, {}^1u] \\
&\to \langle u \| F\rangle[{}^0\tilde{\mu}[x].\langle x \| F\rangle, {}^1u, {}^2u]
\end{aligned}$$

where x_i is a convenient notation to design the variable named with De Bruijn level i (*i.e.* pointers to the i^{th} cell). The exponents ${}^0, {}^1, \ldots$ to number the cells are only there to ease the readability. ⌟

6.4.2 The $\overline{\lambda}_{[lv\tau\star]}$-calculus with De Bruijn levels

We now use De Bruijn levels for variables (and co-variables) that are bounded in the store. We use the mixed notation[13] x_i where the relevant information is x when the variable is bounded within a proof (that is by a λ or $\tilde{\mu}$ binder), and where the relevant information is the number i once the variable has been bounded in the store (at position i). For binders of evaluation contexts, we similarly use De Bruijn levels, but with variables of the form α_i, where, again, α is a fixed name indicating that the variable is binding evaluation contexts, and the relevant information is the index i.

The corresponding syntax is now given by:

$$\begin{array}{llll@{\qquad}llll}
\textbf{Strong values} & v & ::= & \boldsymbol{k} \mid \lambda x_i.t & \textbf{Forcing contexts} & F & ::= & \boldsymbol{\kappa} \mid t \cdot E \\
\textbf{Weak values} & V & ::= & v \mid x_i & \textbf{Catchable contexts} & E & ::= & F \mid \alpha_i \mid \tilde{\mu}[x_i].\langle x_i \| F\rangle\tau \\
\textbf{Terms} & t,u & ::= & V \mid \mu\alpha_i.c & \textbf{Evaluation contexts} & e & ::= & E \mid \tilde{\mu}x_i.c
\end{array}$$

$$\begin{array}{llll}
\textbf{Closures} & l & ::= & c\tau \\
\textbf{Commands} & c & ::= & \langle t \| e\rangle \\
\textbf{Stores} & \tau & ::= & \varepsilon \mid \tau[x_i := t] \mid \tau[\alpha_i := E]
\end{array}$$

The presence of names in the stores is absolutely useless[14] and only there for readability. As the store can be dynamically extended during the execution, the location of a term in the store and the corresponding pointer are likely to evolve (monotonically). Therefore, we need to be able to update De Bruijn levels within terms (contexts, etc...). To this end, we define the lifted term $\uparrow_n^{+i} t$ as the term t where all the free variables x_j with $j > n$ (resp. α_j) have been replaced by x_{j+i}. Formally, they are defined as follows:

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

The corresponding reduction rules are given in Figure 6.8. Note that we choose to perform indices substitutions as soon as they come (maintaining the property that x_n is a variable referring to the $(n+1)^{\text{th}}$ element of the store), while it would also have been possible to store and compose them along

[13]Observe that we could also use usual De Bruijn indices for bounded variables within the terms

[14]In fact, it could even leads to inconsistencies if cell j was of the shape $[x_i := ...]$. The reduction rules will ensure that this never happens but if it was the case, the only relevant information would be the number of the cell (j).

$$\begin{array}{ccll}
\langle t \| \tilde{\mu} x_i.c\rangle\tau & \rightarrow & c[x_n/x_i]\tau[x_n := t] & \text{with } |\tau| = n \\
\langle \mu\alpha_i.c \| E\rangle\tau & \rightarrow & c[\alpha_n/\alpha_i]\tau[\alpha_n := E] & \text{with } |\tau| = n \\
\langle V \| \alpha_n\rangle\tau & \rightarrow & \langle V \| \tau(n)\rangle\tau & \\
\langle x_n \| F\rangle\tau[x_n := t]\tau' & \rightarrow & \langle t \| \tilde{\mu}[x_n].\langle x_n \| F\rangle\tau'\rangle\tau & \\
\langle V \| \tilde{\mu}[x_i].\langle x_i \| F\rangle\tau'\rangle\tau & \rightarrow & \langle V \| \uparrow_n^{+i} F\rangle\tau[x_n := V](\uparrow_n^{+i} \tau') & \text{with } |\tau| = n \\
\langle \lambda x_i.t \| u \cdot E\rangle\tau & \rightarrow & \langle u \| \tilde{\mu} x_n.\langle t[x_n/x_i] \| E\rangle\rangle\tau & \text{with } |\tau| = n
\end{array}$$

Figure 6.8: Reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus with De Bruijn indices

the execution (so that x_n is a variable referring to the $(\sigma(n)+1)^{\text{th}}$ element of the store where σ is the current substitution). This could have seemed more natural for the reader familiar with compilation procedures that do not modify at run time but rather maintain the location of variables through this kind of substitution.

The typing rules are unchanged except for the one where indices should now match the length of the typing context. The resulting type system is given in Figure 6.9.

6.4.3 System F_Υ with De Bruijn levels

The translation for judgments and types, given in Figure 6.11, is almost the same than in the previous section, except that we avoid using names and rather use De Bruijn levels.

As for the target language, it is again an adaptation of System F with stores (lists), in which store subtyping is now witnessed by explicit coercions.

Definition 6.38 (Coercion). We defined coercions to witness store subtyping $\Upsilon' <: \Upsilon$ as finite monotonic functions σ such that $\mathrm{dom}(\sigma) = [\![0, |\Upsilon|-1]\!]$, $\mathtt{codom}(\sigma) \subseteq [\![0, |\Upsilon'|-1]\!]$ and such that for all $i < |\Upsilon|$, $\Upsilon_i = \Upsilon'_{\sigma(i)}$. ⌟

In other words, σ indicates where to find each type of the list Υ in the list Υ'. We denote by $\sigma_{|n}$ the restriction of σ to $[0, n-1]$ and $\mathtt{id}_n$ the identity on $[0, n-1]$. We also define σ_p^+ the canonical extension of a function σ whose domain is $[\![0, n-1]\!]$ for some n and whose co-domain is included in $[\![0, p-1]\!]$ for some p by:

$$\sigma_p^+ : \left\{ \begin{array}{ccc} [0,n] & \rightarrow & [0,p] \\ i < n & \mapsto & \sigma(i) \\ n & \mapsto & p \end{array} \right.$$

Lemma 6.39. *If σ witnesses $\Upsilon' <: \Upsilon$ for some Υ, Υ', then $\sigma^+_{|\Upsilon'|}$ witnesses $\Upsilon', A <: \Upsilon, A$ for any type A.*

As we now got rid of names, we will now split stores with respect to an index. So that if we consider for instance a store of type $\Upsilon' <: (\Upsilon_0, A, \Upsilon_1)$, the knowledge of the position where to find the expected element of type A becomes crucial. In practice, it will be guided by the coercion witnessing $\Upsilon' <: (\Upsilon_0, A, \Upsilon_1)$. But to ensure the correctness of our typing rules, we now need to consider second-order variables (which are in fact vectors of second-order variables) with their arities. That is to say that we should denote by Y^p the vector of variables $Y_0, \ldots, Y_{p-1}$ and that $\forall Y <: \Upsilon.A$ is equivalent

$$\forall p_0 \forall Y^{p_0} \ldots \forall p_n \forall Y^{p_n}.(Y^{p_0}\Upsilon(0)Y^{p_1}\Upsilon(1)\ldots Y^{p_n}) <: \Upsilon \rightarrow A$$

where we have in fact $p_0 = \sigma(0)$, $p_1 = \sigma(1) - p_0 - 1$, etc... In particular, a careful manipulation of variables with their arities allows us to prove the following lemma:

Lemma 6.40. *The typing rules given for coercions in Figure 6.10 are equivalent to Definition 6.38,* i.e. *for all Υ, Υ', for all $i < |\Upsilon|$, $\Upsilon_i = \Upsilon'_{\sigma(i)}$.*

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

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

$$\frac{\Gamma \vdash_V V : A}{\Gamma \vdash_t V : A} \qquad \frac{\Gamma, \alpha_n : A^{\perp\!\!\!\perp} \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_t \mu\alpha_n.c : A} \qquad \frac{\Gamma \vdash_E E : A^{\perp\!\!\!\perp}}{\Gamma \vdash_e E : A^{\perp\!\!\!\perp}} \qquad \frac{\Gamma, x_n : A \vdash_c c \quad |\Gamma| = n}{\Gamma \vdash_e \tilde{\mu}x_n.c : A^{\perp\!\!\!\perp}}$$

$$\frac{\Gamma, x_i : A, \Gamma' \vdash_F F : A^{\perp\!\!\!\perp} \quad \Gamma, x_i : A \vdash_\tau \tau : \Gamma' \quad |\Gamma| = i}{\Gamma \vdash_E \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau : A^{\perp\!\!\!\perp}} \qquad \frac{}{\Gamma \vdash_\tau \varepsilon : \varepsilon}$$

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

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

Figure 6.9: Typing rules for the $\overline{\lambda}_{[l v \tau \star]}$-calculus with De Bruijn

$$\frac{(x : A) \in \Gamma}{\Gamma; \Sigma \vdash x : A}\,(\mathrm{Ax}) \qquad \frac{\Gamma, x : A; \Sigma \vdash t : B \quad |\Gamma| = n}{\Gamma; \Sigma \vdash \lambda x.t : A \to B}\,(\lambda) \qquad \frac{\Gamma; \Sigma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma; \Sigma \vdash t\,u : B}\,(@)$$

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

$$\frac{(c : A) \in \mathcal{S}}{\Gamma; \Sigma \vdash \boldsymbol{k} : A}\,(c) \qquad \frac{\Gamma, x_{\tau_0} : \Upsilon_0, x : A, x_{\tau_1} : \Upsilon_1; \Sigma \vdash t : A \quad \Gamma \vdash \tau : \Upsilon_0, B, \Upsilon_1 \quad |\Upsilon_0| = n}{\Gamma; \Sigma \vdash \mathtt{let}\, x_{\tau_0}, x, x_{\tau_1} = \mathtt{split}\;\; \mathtt{as}\,(\tau)\; \mathtt{in}\;\; n\, \mathtt{in}\, t : A}\,(\mathtt{split})$$

$$\frac{}{\Gamma; \Sigma \vdash \varepsilon : \varepsilon \rhd_\tau \varepsilon}\,(\varepsilon) \qquad \frac{\Gamma; \Sigma \vdash t : \Upsilon \rhd_t A}{\Gamma; \Sigma \vdash [t] : \Upsilon \rhd_\tau A}\,(\tau_t) \qquad \frac{\Gamma; \Sigma \vdash t : \Upsilon \rhd_E A}{\Gamma; \Sigma \vdash [t] : \Upsilon \rhd_\tau A^{\perp\!\!\!\perp}}\,(\tau_E)$$

$$\frac{\Gamma \vdash \tau : \Upsilon_0 \rhd_\tau \Upsilon \qquad \Gamma \vdash \tau' : (\Upsilon_0, \Upsilon) \rhd_\tau \Upsilon'}{\Gamma \vdash \tau\tau' : \Upsilon_0 \rhd_\tau \Upsilon, \Upsilon'}\,(\tau\tau') \qquad \frac{}{\Sigma \vdash \sigma : \Upsilon' \mathrel{<:} \varepsilon}\,(\mathrel{<:}_\varepsilon)$$

$$\frac{(\sigma : \Upsilon' \mathrel{<:} \Upsilon) \in \Sigma}{\Sigma \vdash \sigma : \Upsilon' \mathrel{<:} \Upsilon}\,(\mathrel{<:}_{\mathrm{ax}}) \qquad \frac{\Sigma \vdash \sigma : \Upsilon' \mathrel{<:} \Upsilon \quad \sigma(|\Upsilon|) = |\Upsilon'|}{\Sigma \vdash \sigma : (\Upsilon', A) \mathrel{<:} (\Upsilon, A)}\,(\mathrel{<:}_1) \qquad \frac{\Sigma \vdash \sigma : \Upsilon' \mathrel{<:} \Upsilon}{\Sigma \vdash \sigma : (\Upsilon', A) \mathrel{<:} \Upsilon}\,(\mathrel{<:}_2)$$

Figure 6.10: Typing rules of System F_Υ with De Bruijn levels

$$\begin{array}{lcl}
[\![\Gamma \vdash_e e : A^{\perp\!\!\!\perp}]\!] & \triangleq & \vdash [\![e]\!]_e : [\![\Gamma]\!]_\Gamma \triangleright_e \iota(A) \\
[\![\Gamma \vdash_t t : A]\!] & \triangleq & \vdash [\![t]\!]_t : [\![\Gamma]\!]_\Gamma \triangleright_t \iota(A) \\
[\![\Gamma \vdash_E E : A^{\perp\!\!\!\perp}]\!] & \triangleq & \vdash [\![E]\!]_E : [\![\Gamma]\!]_\Gamma \triangleright_E \iota(A) \\
[\![\Gamma \vdash_V V : A]\!] & \triangleq & \vdash [\![V]\!]_V : [\![\Gamma]\!]_\Gamma \triangleright_V \iota(A) \\
[\![\Gamma \vdash_F F : A^{\perp\!\!\!\perp}]\!] & \triangleq & \vdash [\![F]\!]_F : [\![\Gamma]\!]_\Gamma \triangleright_F \iota(A)
\end{array}
\qquad
\begin{array}{lcl}
[\![\Gamma \vdash_v v : A]\!] & \triangleq & \vdash [\![v]\!]_v : [\![\Gamma]\!]_\Gamma \triangleright_v \iota(A) \\
[\![\Gamma \vdash_c c]\!] & \triangleq & \vdash [\![c]\!]_c : [\![\Gamma]\!]_\Gamma \triangleright_c \bot \\
[\![\Gamma \vdash_l l]\!] & \triangleq & \vdash [\![l]\!]_l^{|\Gamma|} : [\![\Gamma]\!]_\Gamma \triangleright_c \bot \\
[\![\Gamma \vdash_\tau \tau : \Gamma']\!] & \triangleq & \vdash [\![\tau]\!]_\tau : [\![\Gamma]\!]_\Gamma \triangleright_\tau [\![\Gamma']\!]_\Gamma
\end{array}$$

$$[\![\varepsilon]\!]_\Gamma \triangleq \varepsilon \qquad [\![\Gamma, x_i : A]\!]_\Gamma \triangleq [\![\Gamma]\!]_\Gamma, \iota(A) \qquad [\![\Gamma, \alpha_i : A^{\perp\!\!\!\perp}]\!]_\Gamma \triangleq [\![\Gamma]\!]_\Gamma, \iota(A)^{\perp\!\!\!\perp}$$

$$\begin{array}{l}
\Upsilon \triangleright_c A \triangleq \forall Y <: \Upsilon.\, Y \to \bot \\
\Upsilon \triangleright_e A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t A) \to \bot \\
\Upsilon \triangleright_t A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_E A) \to \bot \\
\Upsilon \triangleright_E A \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_V A) \to \bot
\end{array}
\qquad
\begin{array}{ll}
\Upsilon \triangleright_V A & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_F A) \to \bot \\
\Upsilon \triangleright_F A & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_v A) \to \bot \\
\Upsilon \triangleright_v A \to B & \triangleq \forall Y <: \Upsilon.\, Y \to (Y \triangleright_t A) \to (Y \triangleright_E B) \to \bot \\
\Upsilon \triangleright_v X & \triangleq X
\end{array}$$

Figure 6.11: Translation of judgments and types

Even though arities are crucial to ensure the correctness of the definition in Figure 6.10 (in particular to define the relation $\sigma : \Upsilon' <: \Upsilon$ by means of inference rules), to ease the notation we will omit the arity most of the time. We will use the notation $\forall Y^n <: \Upsilon.A$ only when necessary.

The syntax of terms and types is given by:

$$\begin{array}{rcl}
t,u & ::= & x \mid \lambda x.t \mid t\,u \mid \tau \mid \lambda\sigma.t \mid t\,\sigma \\
& \mid & \mathtt{let}\,\tau,x,\tau' = \mathtt{split}\ \tau''\,n\,\mathtt{in}\,t \\
\tau,\tau' & ::= & \varepsilon \mid \tau[t]
\end{array}
\qquad
\begin{array}{rcl}
A,B & ::= & X \mid \bot \mid \Upsilon \triangleright_\tau \Upsilon' \mid A \to B \mid \forall Y <: \Upsilon.\, A \\
\Upsilon,\Upsilon' & ::= & \varepsilon \mid \Upsilon,A \mid \Upsilon,A^{\perp\!\!\!\perp} \mid Y
\end{array}$$

Once again, we will use Υ as a shorthand for typing stores of type $\varepsilon \triangleright_\tau A$. The typing rules are given in Figure 6.10 where the typing contexts are divided in two parts, Γ containing typing hypotheses and Σ the subtyping hypotheses, that are defined by:

$$\Gamma,\Gamma' ::= \varepsilon \mid \Gamma, x : A \qquad\qquad \Sigma,\Sigma' ::= \varepsilon \mid \Sigma, \sigma : (\Upsilon' <: \Upsilon)$$

Now that we gave a computational content to the subtyping relation, some properties that were defined axiomatically in Section 8.3 are now deducible from the characteristics of the coercions σ.

Proposition 6.41. *The subtyping relation $<:$ is an order relation on store types.*

1. *For any Υ, $\Sigma \vdash \mathtt{id}_{|\Upsilon|} : \Upsilon <: \Upsilon$*
2. *If $\Sigma \vdash \sigma : \Upsilon <: \Upsilon'$ and $\Sigma \vdash \sigma' : \Upsilon' <: \Upsilon''$, then $\Sigma \vdash \sigma' \circ \sigma : \Upsilon <: \Upsilon''$.*
3. *If $\Sigma \vdash \sigma : \Upsilon <: \Upsilon'$ and $\Sigma \vdash \sigma' : \Upsilon' <: \Upsilon$, then $\sigma' \circ \sigma = \sigma' \circ \sigma = \mathtt{id}_{|\Upsilon|}$ and $\Upsilon = \Upsilon'$.*

Proof. Straightforward from the definition of $\sigma : \Upsilon' <: \Upsilon$:

1. Obvious.
2. For all $i < |\Upsilon|$, we have $\Upsilon''_{\sigma'(\sigma(i)} = \Upsilon'_{\sigma(i))} = \Upsilon_i$.
3. Using the second item, we deduce that $\sigma' \circ \sigma$ witnesses $\Upsilon <: \Upsilon$. Both σ and σ' being monotonic functions, we deduce that $\sigma' = \sigma = \mathtt{id}_{|\Upsilon|}$ and that for all $i < |\Upsilon|$, $\Upsilon_i = \Upsilon'_i$. □

Proposition 6.42. *For any function σ and any types Υ, Υ', if $\vdash \sigma : \Upsilon' <: \Upsilon$ and Υ is of the form $\Upsilon = \Upsilon_0, A, \Upsilon_1$, then Υ' is of the form $\Upsilon' = \Upsilon_0', A, \Upsilon_1'$ such that $|\Upsilon_0'| = \sigma(|\Upsilon_0|)$ and $|\Upsilon_1'| = \sigma(|\Upsilon|) - |\Upsilon_0'| - 1$.*

Proof. Straightforward from the definitions. □

The former propositions shows that the following subtyping rules (where we use a compact version of the second-order variable) are admissible:

$$\frac{\Sigma \vdash \sigma : \Upsilon <: \Upsilon' \quad \Sigma \vdash \sigma' : \Upsilon' <: \Upsilon''}{\Sigma \vdash \sigma' \circ \sigma : \Upsilon <: \Upsilon''}\ (<:_3) \qquad \frac{\Gamma'; \Sigma' \vdash t : B \quad \Sigma \vdash \sigma : Y <: \Upsilon_0, A, \Upsilon_1}{\Gamma; \Sigma \vdash t : B}\ (<:_{\texttt{split}})$$

where $\Gamma' = \Gamma[(Y_0^{\sigma(n)}, A, Y_1)/Y]$, $\Sigma' = \Sigma[(Y_0^{\sigma(n)}, A, Y_1)/X]$, and $Y_0^{\sigma(n)}, Y_1$ are fresh variables. Observe that the second one is a tautology that we only used to avoid the heavy syntactical manipulation of vectors of variables within proof trees.

Lemma 6.43 (Weakening). *The following rules are admissible:*

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

Proof. Easy induction on typing derivations. In the case of second-order quantification, we might need to rename the second-order variable X if it occurs in Σ' (resp.Γ') and not in Σ (resp. Γ). □

6.4.4 A typed CPS translation with De Bruijn levels

We shall now present the translation of terms and prove its correctness with respect to types. The translation, which is given in Figure 6.12, is similar to the translation with names in Section 8.3 plus the manipulation of coercions. Once again, we assume that for each constant $\boldsymbol{k}$ of type A (resp. co-constant $\boldsymbol{\kappa}$ of type $A^{\perp\!\!\perp}$) of the source system, we have a constant of type A in the signature of the target language that we also denote by $\boldsymbol{k}$ (resp. $\boldsymbol{\kappa}$ of type $A \to \bot$). We will now prove a bunch of lemmas that will be useful in the proof of the main theorem.

First, we show that the type of the store expected through the translation can be weakened. This is a sanity-check reflecting the usual weakening in the source language.

Lemma 6.44. *The following rule is admissible for any level o of the hierarchy e, t, E, V, F, v:*

$$\frac{\Gamma; \Sigma \vdash t : \Upsilon \triangleright_o A}{\Gamma; \Sigma \vdash t : \Upsilon, B \triangleright_o A}$$

Proof. Directly follows from the observation that we can always derive:

$$\frac{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon, B}{\Sigma \vdash \sigma : \Upsilon' <: \Upsilon}$$

□

Then we show that the bounded quantification can be composed with subtyping relation witnessed by a coercion, by means of a lifting on the term accordingly with the coercion.

Lemma 6.45. *The following rules is admissible:*

$$\frac{\Gamma; \Sigma \vdash t : \forall Y <: \Upsilon_0 . A \quad \Sigma \vdash \sigma : \Upsilon_1 <: \Upsilon_0}{\Gamma; \Sigma \vdash (\uparrow^\sigma t) : \forall Y <: \Upsilon_1 . A}$$

$$
\begin{array}{ll}
(\uparrow^{\sigma} t)\ \sigma' & \triangleq\ t\ (\sigma' \circ \sigma) \\
(\uparrow^{\sigma} \tau[t]) & \triangleq\ (\uparrow^{\sigma} \tau)[\uparrow^{\sigma} t] \\
 & \\
\llbracket \boldsymbol{k} \rrbracket_v & \triangleq\ \boldsymbol{k} \\
\llbracket \lambda x_i . t \rrbracket_v\ \sigma\ \tau\ u\ E & \triangleq\ \llbracket t \rrbracket_t\ \sigma^{+}_{|\tau|}\ \tau[u]\ E \\
 & \\
\llbracket \boldsymbol{\kappa} \rrbracket_F & \triangleq\ \boldsymbol{\kappa} \\
\llbracket t \cdot E \rrbracket_F\ \sigma\ \tau\ v & \triangleq\ v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket t \rrbracket_t)\ (\uparrow^{\sigma} \llbracket E \rrbracket_E) \\
 & \\
\llbracket v \rrbracket_V\ \sigma\ \tau\ F & \triangleq\ F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket v \rrbracket_v) \\
\llbracket x_i \rrbracket_V\ \sigma\ \tau[t]\tau'\ F & \triangleq\ t\ \mathtt{id}_{|\tau|}\ \tau\ (\lambda \sigma' \tau'' \lambda V. V\ \tau''[\uparrow^{t} V](\uparrow^{\sigma''} \tau')\ (\uparrow^{\sigma''} F)) \\
 & \text{where } n = |\tau| = \sigma(i),\ k = |\tau''| - n,\ p = n + |\tau'|,\ \sigma'' = \sigma' \circ \delta^{+k}_{[n,p]} \\
 & \text{ and } \uparrow^{t} V = \lambda \sigma \tau E. E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} V) \\
 & \\
\llbracket \alpha_i \rrbracket_E\ \sigma\ \tau\ V & \triangleq\ \mathtt{let}\ \tau', x, \tau'' = \mathtt{split}\ \ \mathtt{as}\,(\sigma)\ \mathtt{in}\,(i)\,\tau\ \mathtt{in}\, x\ \mathtt{id}_{|\tau|}\ \tau\ V \\
\llbracket \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau' \rrbracket_E\ \sigma\ \tau\ V & \triangleq\ V\ \mathtt{id}_{|\tau|}\ \tau[\uparrow^{t} V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)\ (\uparrow^{\sigma'} \llbracket F \rrbracket_F) \\
 & \text{where } n = |\tau|,\ k = n - i,\ p = n + |\tau'|,\ \sigma' = \sigma \circ \delta^{+k}_{[i,p]} \\
 & \\
\llbracket V \rrbracket_t\ \sigma\ \tau\ E & \triangleq\ E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket V \rrbracket_V) \\
\llbracket \mu \alpha_i . c \rrbracket_t\ \sigma\ \tau\ E & \triangleq\ \llbracket c \rrbracket_c\ \sigma^{+}_{|\tau|}\ \tau[E] \\
 & \\
\llbracket E \rrbracket_e\ \sigma\ \tau\ t & \triangleq\ t\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma} \llbracket E \rrbracket_E) \\
\llbracket \tilde{\mu} x_i . c \rrbracket_e\ \sigma\ \tau\ t & \triangleq\ \llbracket c \rrbracket_c\ \sigma^{+}_{|\tau|}\ \tau[t] \\
 & \\
\llbracket \langle t \| e \rangle \rrbracket_c\ \sigma\ \tau & \triangleq\ \llbracket e \rrbracket_e\ \sigma\ \tau\ (\uparrow^{\sigma} \llbracket t \rrbracket_t) \\
\llbracket c \tau \rrbracket^{n}_{l}\ \sigma\ \tau' & \triangleq\ \llbracket c \rrbracket_c\ \sigma'\ \tau'(\uparrow^{\sigma'} \llbracket \tau \rrbracket_\tau) \\
 & \text{where } k = |\tau'| - n,\ p = n + |\tau|,\ \sigma' = \sigma \circ \delta^{+k}_{[n,p]} \\
 & \\
\llbracket \varepsilon \rrbracket_\tau & \triangleq\ \varepsilon \\
\llbracket \tau_0[x_i := t] \rrbracket_\tau & \triangleq\ \llbracket \tau_0 \rrbracket_\tau [\llbracket t \rrbracket_t] \\
\llbracket \tau_0[\alpha_i := E] \rrbracket_\tau & \triangleq\ \llbracket \tau_0 \rrbracket_\tau [\llbracket E \rrbracket_E] \\
 & \\
\delta^{+i}_{[n,p]} & \triangleq\ \begin{cases} j \mapsto j + i & \text{if } n \leq j < p \\ j \mapsto j & \text{if } j < n \end{cases}
\end{array}
$$

Figure 6.12: Translation of terms

Proof. We assume that the variable X is not $FV(\Gamma,\Sigma)$, otherwise it suffices to rename it. Unfolding the definition of $\uparrow^{\sigma} t$, we can derive:

$$\dfrac{\dfrac{\dfrac{\Gamma;\Sigma\vdash t:\forall X<:\Upsilon_0.A}{\Gamma;\Sigma,\sigma':X<:\Upsilon_1\vdash t:\forall X<:\Upsilon_0.A}\qquad\dfrac{\Sigma\vdash\sigma:\Upsilon'<:\Upsilon_1\quad\Sigma,\sigma':X<:\Upsilon_1\vdash\sigma':X<:\Upsilon_1}{\Sigma,\sigma':X<:\Upsilon_1\vdash\sigma'\circ\sigma:X<:\Upsilon_0}}{\Gamma;\Sigma,\sigma':X<:\Upsilon_1\vdash t\ (\sigma'\circ\sigma):A}\qquad X\notin FV(\Gamma,\Sigma)}{\Gamma;\Sigma\vdash\lambda\sigma'.t\ (\sigma'\circ\sigma):\forall X<:\Upsilon_1.A}$$

where we use Lemma 6.43 to weaken $\Sigma,\sigma:X<:\Upsilon_1$. □

We deduce from the former lemma the following corollary that will be crucial when typing the translation of terms.

Corollary 6.46. *For any level o of the hierarchy e,t,E,V,F,v, the following rule are admissible:*

$$\dfrac{\Gamma;\Sigma\vdash t:\Upsilon_0\triangleright_o A\quad\Sigma\vdash\sigma:\Upsilon_1<:\Upsilon_0}{\Gamma;\Sigma\vdash(\uparrow^{\sigma}t):\Upsilon_1\triangleright_o A}\qquad\qquad\dfrac{\Gamma;\Sigma\vdash\tau:\Upsilon_0\triangleright_\tau\Upsilon\quad\Sigma\vdash\sigma:\Upsilon_1\Upsilon<:\Upsilon_0\Upsilon}{\Gamma;\Sigma\vdash(\uparrow^{\sigma}\tau):\Upsilon_1\triangleright_\tau\Upsilon}$$

The following lemma shows that the operation of lifting values to terms is sound with respect to the translation of types.

Lemma 6.47 (Lifting values). *The following rule is admissible:*

$$\dfrac{\Gamma;\Sigma\vdash V:\Upsilon\triangleright_V A}{\Gamma;\Sigma\vdash\uparrow^{t}V:\Upsilon\triangleright_t A}\ (\uparrow)$$

Proof.

$$\dfrac{\dfrac{\Pi_E\qquad\dfrac{\Gamma;\Sigma\vdash V:\Upsilon\triangleright_V A\quad\dfrac{}{\sigma:Y<:\Upsilon\vdash\sigma:Y<:\Upsilon}\,(<:_{\mathrm{ax}})}{\Gamma;\Sigma,\sigma:Y<:\Upsilon\vdash\uparrow^{\sigma}V:Y\triangleright_V A}}{\Gamma,\tau:Y,E:\Upsilon\triangleright_E A;\Sigma;\sigma:Y<:\Upsilon\vdash E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma}V):\bot}\,(@)}{\Gamma;\Sigma\vdash\lambda\sigma\tau E.E\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^{\sigma}V):\Upsilon\triangleright_t A}\,(\lambda)$$

where we used Corollary 6.46 and Π_E is the following derivation:

$$\dfrac{\dfrac{\dfrac{}{E:\Upsilon\triangleright_E A;\vdash E:Y\triangleright_E A\to\bot}\,(\mathrm{Ax})\quad\dfrac{}{\vdash\mathtt{id}_{|\tau|}:Y<:Y}\,(<:_{\mathrm{ax}})}{E:\Upsilon\triangleright_E A;\vdash E\ \mathtt{id}_{|\tau|}:Y\to Y\triangleright_V A\to\bot}\,(\forall_E)\qquad\dfrac{}{\tau:Y;\vdash\tau:Y}\,(\mathrm{Ax})}{\tau:Y,E:\Upsilon\triangleright_E A;\vdash E\ \mathtt{id}_{|\tau|}\ \tau:Y\triangleright_V A\to\bot}\,(@)$$

□

We now prove the soundness of the rules for forming stores through the translation.

Lemma 6.48 (Store formation). *The following rules are admissible:*

$$\dfrac{\Gamma;\Sigma\vdash\tau:\Upsilon\quad\Gamma;\Sigma\vdash t:\Upsilon\triangleright_t A}{\Gamma;\Sigma\vdash\tau[t]:\Upsilon,A}\qquad\qquad\dfrac{\Sigma\vdash\sigma:\Upsilon<:[\![\Gamma_0]\!]}{\Sigma\vdash\sigma^{+}_{|\Upsilon|}:(\Upsilon,A)<:[\![\Gamma_0,A]\!]}$$

The same holds for $\Gamma\vdash E:\Upsilon\triangleright_E\iota(A)$ and $\Gamma\vdash\tau[E]:\Upsilon,A^{\perp\!\!\perp}$.

Proof. The left rule is a straightforward application of $(\tau\tau')$- and (τ_t)-rules:

$$\dfrac{\Gamma;\Sigma\vdash\tau[t]:Y,A\qquad\dfrac{\Gamma;\Sigma\vdash t:Y\triangleright_t\iota(A)}{\Gamma;\Sigma\vdash[t]:Y\triangleright_\tau\iota(A)^{\perp\!\!\perp}}\,(\tau_t)}{\Gamma;\Sigma\vdash\tau[t]:Y,A}\,(\tau\tau')$$

The right one is a reformulation of Lemma 6.39. □

Similarly, we can prove that the shifts accordingly to a coercion are sound with respect to types:

Lemma 6.49 (Shifts). *For any $\Upsilon_0, \Upsilon_0', \Upsilon_1$, if $\sigma : \Upsilon_0' <: \Upsilon_0$ and $n = |\Upsilon_0|, p = n + |\Upsilon_1|$, $k = [\Upsilon_0'| - |\Upsilon_0|$, if we define $\sigma' = \sigma \circ \delta^{+k}_{[n,p]}$ then $\sigma' : (\Upsilon_0'\Upsilon_1) <: (\Upsilon_0\Upsilon_1)$.*
In particular, the following rules are admissible for any level o:

$$\frac{\Gamma;\Sigma \vdash t : \Upsilon_0\Upsilon_1 \triangleright_o A \quad \Sigma \vdash \sigma : \Upsilon_0' <: \Upsilon_0}{\Gamma;\Sigma \vdash (\uparrow^{\sigma'} t) : \Upsilon_0'\Upsilon_1 \triangleright_o A} \qquad \frac{\Gamma;\Sigma \vdash \tau : \Upsilon_0 \triangleright_\tau \Upsilon_1 \quad \Sigma \vdash \sigma : \Upsilon_0' <: \Upsilon_0}{\Gamma;\Sigma \vdash (\uparrow^{\sigma'} \tau) : \Upsilon_0' \triangleright_\tau \Upsilon_1}$$

Proof. We denote by $\Upsilon(i)$ the i^{th}-element of the list Υ. By definition, we have:

$$\sigma'(i) = \begin{cases} i + k & \text{if } n \leq i < p \\ \sigma'(i) & \text{if } j < n \end{cases}$$

We have:

$$\begin{array}{rcll} (\Upsilon_0'\Upsilon_1)(\sigma'(i)) & = & \Upsilon_0'(\sigma'(i)) = \Upsilon_0'(\sigma(i)) = \Upsilon_0(i) & \text{(if } i < n) \\ (\Upsilon_0'\Upsilon_1)(\sigma'(i)) & = & (\Upsilon_0'\Upsilon_1)(i + k) = \Upsilon_1(i + k - |\Upsilon_0'|) = \Upsilon_1(i - |\Upsilon_0|) = (\Upsilon_0\Upsilon_1)(i) & \text{(otherwise)} \end{array}$$

Thus we can conlude $\sigma' : (\Upsilon_0'\Upsilon_1) <: (\Upsilon_0\Upsilon_1)$. □

We are finally equipped to prove the main theorem of this section, that is the correctness of the translation with respect to types.

Theorem 6.50. *The translation is well-typed, i.e.*

1. *if* $\Gamma \vdash_v v : A$ *then* $\llbracket \Gamma \vdash_v v : A \rrbracket$
2. *if* $\Gamma \vdash_F F : A^{⫫}$ *then* $\llbracket \Gamma \vdash_F F : A^{⫫} \rrbracket$
3. *if* $\Gamma \vdash_V V : A$ *then* $\llbracket \Gamma \vdash_V V : A \rrbracket$
4. *if* $\Gamma \vdash_E E : A^{⫫}$ *then* $\llbracket \Gamma \vdash_E E : A^{⫫} \rrbracket$
5. *if* $\Gamma \vdash_t t : A$ *then* $\llbracket \Gamma \vdash_t t : A \rrbracket$
6. *if* $\Gamma \vdash_e e : A^{⫫}$ *then* $\llbracket \Gamma \vdash_e e : A^{⫫} \rrbracket$
7. *if* $\Gamma \vdash_c c$ *then* $\llbracket \Gamma \vdash_c c \rrbracket$
8. *if* $\Gamma \vdash_l l$ *then* $\llbracket \Gamma \vdash_l l \rrbracket$
9. *if* $\Gamma \vdash_\tau \tau$ *then* $\llbracket \Gamma \vdash_\tau \tau : \Gamma' \rrbracket$

Proof. The proof is almost the same as the proof of Theorem 6.33, using the previous lemmas. We reason by induction over the typing rules of Figure 6.9. We (ab)use of Lemma 6.43 to make the derivations more compact by systematically weakening contexts as soon as possible, and compact the first $(\forall_I)$ and (λ) rules in one rule.

1. Strong values

- **Case** $\llbracket \boldsymbol{k} \rrbracket_v$. $\llbracket \boldsymbol{k} \rrbracket_v = \boldsymbol{k}$, which has the desired type by hypothesis.

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

$$\frac{\Gamma, x_i : A \vdash_t t : B \quad |\Gamma| = i}{\Gamma \vdash_v \lambda x_i : A \to B}$$

Hence, we get by induction a proof Π_t of $\llbracket t \rrbracket_t : \llbracket \Gamma, x_i : A \rrbracket \triangleright_t \iota(B)$ and we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_t}{\vdash \llbracket t \rrbracket_t : \forall Y' <: \llbracket \Gamma, x_i : A \rrbracket . Y' \to Y' \triangleright_E \iota(B) \to \bot} \quad \boldsymbol{\Pi_\sigma}}{;\sigma : Y <: \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket_t\ \sigma^+_{|\tau|} : (Y, \iota(A)) \to (Y, \iota(A)) \triangleright_E \iota(B) \to \bot}\,(\forall_E) \quad \boldsymbol{\Pi_\tau}}{\tau : Y, u : Y \triangleright_t \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket_t\ \sigma^+_{|\tau|}\ \tau[u] : (Y, \iota(A)) \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{\tau : Y, u : Y \triangleright_t \iota(A), E : Y \triangleright_E \iota(B); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket_t\ \sigma^+_{|\tau|}\ \tau[u]\ E^+ : \bot}\,(@)}{\vdash \lambda\sigma\tau u E. \llbracket t \rrbracket_t\ \sigma^+_{|\tau|}\ \tau[u]\ E : \forall Y <: \llbracket \Gamma \rrbracket . Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\,(\lambda)$$

where:

- Π_E is a proof of $E : Y \triangleright_E \iota(B) \vdash E : (Y, \iota(A)) \triangleright_E \iota(B)$ (derivable according to Lemma 6.44);
- Π_τ is a proof of $\tau : Y, u : Y \triangleright_t \iota(A); \vdash \tau[u] : Y, \iota(A)$ (derivable according to Lemma 6.48);
- Π_σ is obtained by Lemma 6.48:

$$\dfrac{\dfrac{}{\sigma : Y <: \llbracket\Gamma\rrbracket \vdash \sigma : Y <: \llbracket\Gamma\rrbracket}\,(<:_{\mathrm{ax}})}{\sigma : Y <: \llbracket\Gamma\rrbracket \vdash \sigma^{+}_{|\tau|} : (Y, \iota(A)) <: \llbracket\Gamma, x_i : A\rrbracket}$$

2. Forcing contexts

- **Case** $\llbracket\boldsymbol{\kappa}\rrbracket_F$. $\llbracket\boldsymbol{\kappa}\rrbracket_F = \boldsymbol{\kappa}$, which has the desired type by hypothesis.

- **Case** $\llbracket t.E\rrbracket_F$. In the source language, we have:

$$\dfrac{\Gamma \vdash_t t : A \quad \Gamma \vdash_E E : B^{\perp\!\!\perp}}{\Gamma \vdash_F t \cdot E : (A \to B)^{\perp\!\!\perp}}$$

Therefore we have by induction hypothesis that $\vdash \llbracket t\rrbracket_t : \llbracket\Gamma\rrbracket_\Gamma \triangleright_t \iota(A)$ and $\vdash \llbracket E\rrbracket_t : \llbracket\Gamma\rrbracket_\Gamma \triangleright_E \iota(B)$, so that we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : Y \triangleright_v \iota(A) \to \iota(B); \vdash v : \forall Y' <: Y : Y' \to Y' \triangleright_t \iota(A) \to Y' \triangleright_E \iota(B) \to \bot}\,(\mathrm{Ax}) \quad \boldsymbol{\Pi_\sigma}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \vdash v\ \mathtt{id}_{|\tau|} : Y \to Y \triangleright_t \iota(A) \to Y \triangleright_E B \to \bot}\,(\forall_I) \quad \boldsymbol{\Pi_\tau}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ : Y \triangleright_t \iota(A) \to Y \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_t}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \sigma : Y <: \llbracket\Gamma\rrbracket \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t\rrbracket_t) : Y \triangleright_E \iota(B) \to \bot}\,(@) \quad \boldsymbol{\Pi_E}}{\tau : Y, v : Y \triangleright_v \iota(A) \to \iota(B); \sigma : Y <: \llbracket\Gamma\rrbracket \vdash v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t\rrbracket_t)\ (\uparrow^\sigma \llbracket E\rrbracket_E) : \bot}\,(@)}{\vdash \lambda\sigma\tau v.v\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket t\rrbracket_t)\ (\uparrow^\sigma \llbracket E\rrbracket_E) : \forall Y <: \llbracket\Gamma\rrbracket_\Gamma . Y \to Y \triangleright_v \iota(A) \to \iota(B) \to \bot}\,(\lambda)$$

where:

- Π_E is a proof of $\varepsilon; \sigma : Y <: \llbracket\Gamma\rrbracket \vdash (\uparrow^\sigma \llbracket E\rrbracket_E) : Y \triangleright_E \iota(B)$, derived from the induction hypothesis for t and Corollary 6.46;
- Π_t is a proof of $\varepsilon; \sigma : Y <: \llbracket\Gamma\rrbracket \vdash (\uparrow^\sigma \llbracket t\rrbracket_t) : Y \triangleright_t \iota(A)$, derived from the induction hypothesis for E and Corollary 6.46;
- Π_τ is the axiom rule $\tau : Y; \vdash \tau : Y$;
- Π_σ is a proof of $\mathtt{id}_{|\tau|} : Y <: Y$ (Proposition 6.41).

3. Weak values

- **Case** $\llbracket v \rrbracket_V$. In the source language, we have:

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

Hence we have by induction hypothesis that $\vdash \llbracket v \rrbracket_v : \llbracket \Gamma \rrbracket_\Gamma \triangleright_v \iota(A)$ and we can derive:

$$\begin{prooftree}
\AxiomC{$F : Y \triangleright_F \iota(A) \vdash F : \forall Y' <: Y.Y' \to Y' \triangleright_v \iota(A) \to \bot$}
\AxiomC{Π_Y}
\RightLabel{(@)}
\BinaryInfC{$F : Y \triangleright_F \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash F\ \mathtt{id}_{|\tau|} : Y \to Y \triangleright_v \iota(A) \to \bot$}
\AxiomC{}
\UnaryInfC{$\tau : Y; \vdash \tau : Y$}
\RightLabel{(@)}
\BinaryInfC{$\tau : Y, F : Y \triangleright_F \iota(A) \vdash F\ \mathtt{id}_{|\tau|}\ \tau : Y \triangleright_v \iota(A) \to \bot$}
\AxiomC{Π_v}
\RightLabel{(@)}
\BinaryInfC{$\tau : Y, F : Y \triangleright_F \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket \vdash F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket v \rrbracket_v) : \bot$}
\RightLabel{(λ)}
\UnaryInfC{$\vdash \lambda \sigma \tau F.F\ \mathtt{id}_{|\tau|}\ \tau\ (\uparrow^\sigma \llbracket v \rrbracket_v) : \forall Y <: \llbracket \Gamma \rrbracket.Y \to Y \triangleright_F \iota(A) \to \bot$}
\end{prooftree}$$

where:

- Π_v is a proof of $\varepsilon; \sigma : Y <: \llbracket \Gamma \rrbracket \vdash (\uparrow^\sigma \llbracket v \rrbracket_v) : Y \triangleright_v \iota(A)$, derivable from the induction hypothesis and Corollary 6.46.
- Π_τ is the axiom rule $\tau : Y; \vdash \tau : Y$
- Π_Y is a proof of $\mathtt{id}_{|\tau|} : Y <: Y$ (Proposition 6.41)

- **Case** $\llbracket x_i \rrbracket_V$. In the source language, we have:

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

so that Γ is of the form $\Gamma', x_i : A, \Gamma''$. By definition, we have:

$$\llbracket x_i \rrbracket_V = \lambda \sigma \tau F.\,\mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ n\,\tau\ \mathtt{in}\,t\ \mathtt{id}_n\ \tau_0\ (\lambda \sigma' \tau_0' \lambda V.V\ \tau''[\uparrow^t\ V](\uparrow^{\sigma''} \tau_1)\ (\uparrow^{\sigma''} F))$$

where $n = \sigma(i)$,$k = |\tau_0| - n$, $p = n + |\tau_1|$, $\sigma'' = \sigma' \circ \delta^{+k}_{[n,p]}$.

$$\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$t : Y_0^n \triangleright_t \iota(A) \vdash t : Y_0^n \triangleright_t \iota(A)$}
\AxiomC{}
\RightLabel{($<:_{\text{ax}}$)}
\UnaryInfC{$\vdash \mathtt{id}_n : Y_0^n <: Y_0^n$}
\RightLabel{($\forall_E$)}
\BinaryInfC{$t : Y_0^n \triangleright_t \iota(A); \vdash t\ \mathtt{id}_n : Y_0^n \to Y_0^n \triangleright_E \iota(A) \to \bot$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$\tau_0 : Y_0^n \vdash \tau_0 : Y_0^n$}
\RightLabel{(@)}
\BinaryInfC{$\tau_0 : Y_0^n, t : Y_0^n \triangleright_t A; \vdash t\ \mathtt{id}_n\ \tau_0 : Y_0^n \triangleright_E \iota(A) \to \bot$}
\AxiomC{$\mathbf{\Pi}_E$}
\RightLabel{(@)}
\BinaryInfC{$\tau_0 : Y_0^n, t : Y_0^n \triangleright_t \iota(A), \tau_1 : (Y_0^n, n : \iota(A)) \triangleright_\tau Y_1, F : (Y_0^n, n : \iota(A), Y_1) \triangleright_F \iota(A); \vdash t\ \mathtt{id}_n\ \tau_0\ E : \bot$}
\AxiomC{$|Y_0^n| = n$}
\RightLabel{(split)}
\BinaryInfC{$\tau : (Y_0^n, n : \iota(A), Y_1), F : (Y_0^n, n : \iota(A), Y_1) \triangleright_F \iota(A); \vdash \mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \mathtt{id}_n\ \tau_0\ E : \bot$}
\AxiomC{$\mathbf{\Pi}_\sigma$}
\RightLabel{($<:_{\mathtt{split}}$)}
\BinaryInfC{$\tau : Y, F : Y \triangleright_F \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \mathtt{id}_n\ \tau_0\ E : \bot$}
\RightLabel{(λ)}
\UnaryInfC{$\vdash \lambda \sigma \tau F.\,\mathtt{let}\,\tau_0, t, \tau_1 = \mathtt{split}\ \tau\, n\, \mathtt{in}\, t\ \mathtt{id}_n\ \tau_0\ E : \forall Y <: \llbracket \Gamma \rrbracket_\Gamma.Y \to Y \triangleright_F \iota(A) \to \bot$}
\end{prooftree}$$

where:

- Π_σ is simply the axiom rule:

$$\begin{prooftree}
\AxiomC{}
\RightLabel{($<:_{\text{ax}}$)}
\UnaryInfC{$\sigma : Y <: (\llbracket \Gamma_0 \rrbracket_\Gamma, n : \iota(A), \llbracket \Gamma_1 \rrbracket_\Gamma) \vdash \sigma : Y <: (\llbracket \Gamma_0 \rrbracket_\Gamma, n : \iota(A), \llbracket \Gamma_1 \rrbracket_\Gamma)$}
\end{prooftree}$$

- $E = \lambda \sigma' \tau'' \lambda V.V\ \tau_0'[\uparrow^t\ V](\uparrow^{\sigma''} \tau_1)\ (\uparrow^{\sigma''} F))$ and Π_E is the following derivation:

$$\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$V : Y_0' \triangleright_V \iota(A); \vdash \uparrow^t V : Y_0' \triangleright_t \iota(A)$}
\AxiomC{}
\UnaryInfC{$\vdash \mathtt{id}_p : Y_0', A, Y_1 <: Y_0', A, Y_1$}
\RightLabel{($\forall_E$)}
\BinaryInfC{$V : Y_0' \triangleright_V \iota(A); \vdash V\ \mathtt{id}_p : (Y_0', \iota(A), Y_1) \to (Y_0', \iota(A), Y_1) \triangleright_F \iota(A) \to \bot$}
\AxiomC{$\mathbf{\Pi}_\tau$}
\RightLabel{(@)}
\BinaryInfC{$\tau_1 : (Y_0^n, \iota(A)) \triangleright_\tau Y_1, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A); \vdash V\ \mathtt{id}_p\ \tau_0'[\uparrow^t V](\uparrow^{\sigma'} \tau_1) : (Y_0', \iota(A), Y_1) \triangleright_F \iota(A) \to \bot$}
\AxiomC{$\mathbf{\Pi}_F$}
\RightLabel{(@)}
\BinaryInfC{$\Gamma, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A); \sigma' : Y_0' <: Y_0^n \vdash V\ \mathtt{id}_p\ \tau_0'[\uparrow^t V](\uparrow^{\sigma''} \tau')\ (\uparrow^{\sigma''} F) : \bot$}
\RightLabel{(λ)}
\UnaryInfC{$\Gamma \vdash \lambda \sigma' \tau_0' V.V\ \mathtt{id}_p\ \tau_0'[\uparrow^t V](\uparrow^{\sigma''} \tau_1)\ (\uparrow^{\sigma''} F) : Y_0^n \triangleright_F \iota(A)$}
\end{prooftree}$$

where $\Gamma = \tau_1 : (Y_0^n, \iota(A)) \triangleright_\tau Y_1, F : (Y_0^n, \iota(A), Y_1) \triangleright_F \iota(A)$.

- Π_F is the following proof, obtained by Lemma 6.49:

$$\dfrac{\overline{F : (Y_0^n, \iota(A), Y_1) \triangleright_F \iota(A); \vdash F : (Y_0^n, \iota(A), Y_1) \triangleright_F \iota(A)}\;(\text{Ax}) \qquad \overline{\sigma' : Y_0' <: Y_0^n \vdash \sigma' : Y_0' <: Y_0^n}\;(\text{Ax})}{F : (Y_0^n, \iota(A), Y_1) \triangleright_F \iota(A); \sigma_1 : Y_0' <: Y_0^n \vdash (\uparrow^{\sigma''} F) : (Y_0', \iota(A), Y_1) \triangleright_F \iota(A)}$$

- Π_τ is the following derivation

$$\dfrac{\dfrac{\overline{\tau_0' : Y_0' \vdash \tau_0' : Y_0'}\;(\text{Ax}) \qquad \dfrac{\dfrac{\overline{V : Y_0' \triangleright_V \iota(A) \vdash V : Y_0' \triangleright_V A}\;(\text{Ax})}{V : Y_0' \triangleright_V \iota(A) \vdash \uparrow^t V : Y_0' \triangleright_t A}\;(\uparrow)}{V : Y_0' \triangleright_V \iota(A) \vdash [\uparrow^t V] : Y_0' \triangleright_\tau \iota(A)}\;(\tau_t)}{Y_0' <: Y_0^n, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[\uparrow^t V] : Y_0', n : \iota(A)}\;(\tau\tau') \qquad \mathbf{\Pi}_{\tau_1}}{\tau_1 : (Y_0^n, \iota(A)) \triangleright_\tau Y_1, Y_0' <: Y_0^n, \tau_0' : Y_0', V : Y_0' \triangleright_V \iota(A) \vdash \tau_0'[V](\uparrow^{\sigma''} \tau_1) : Y_0', \iota(A), Y_1}\;(\tau_{<:})$$

- Π_{τ_1} is obtained by Lemma 6.49:

$$\dfrac{\overline{\tau_1 : (Y_0, n : \iota(A)) \triangleright_\tau Y_1 \vdash \tau_1 : (Y_0, \iota(A)) \triangleright_\tau Y_1}\;(\text{Ax}) \qquad \overline{\sigma' : Y_0' <: Y_0^n \vdash \sigma' : Y_0' <: Y_0^n}\;(<:_{\text{ax}})}{\tau_1 : (Y_0^n, \iota(A)) \triangleright_\tau Y_1; \sigma' : Y_0' <: Y_0 \vdash (\uparrow^{\sigma''} \tau_1) : Y_0', n : \iota(A) \triangleright_\tau Y_1}$$

4. Catchable contexts

- **Case** $\llbracket F \rrbracket_E$. This case is similar to the case $\llbracket v \rrbracket_V$.

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

$$\dfrac{\Gamma, x_i : A, \Gamma' \vdash_F F : A^{\perp\!\!\perp} \quad \Gamma, x_i : A \vdash_\tau \tau' : \Gamma' \quad |\Gamma| = i}{\Gamma \vdash_E \tilde{\mu}[x_i].\langle x_i \| F \rangle \tau' : A^{\perp\!\!\perp}}$$

We have by induction hypothesis a proof of $\vdash \llbracket \tau' \rrbracket_\tau : \llbracket \Gamma, x_i : A \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma$ and a proof Π_F of $\vdash \llbracket F \rrbracket_F :$ $\llbracket \Gamma, x_i : A, \Gamma' \rrbracket_\Gamma \triangleright_F \iota(A)$. We can thus derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{V : Y \triangleright_V \iota(A); \vdash V : Y \triangleright_t \iota(A)}\;(\text{Ax}) \qquad \overline{\vdash \mathtt{id}_p : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) <: Y}}{V : Y \triangleright_V \iota(A); \vdash V\ \mathtt{id}_p : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \to (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A) \to \bot}\;(\forall_E) \qquad \mathbf{\Pi}_\tau}{\tau : Y, V : Y \triangleright_V \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash V\ \mathtt{id}_p\ \tau[\uparrow^t V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau) : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A) \to \bot}\;(@) \qquad \mathbf{\Pi}_F}{\Gamma, \tau : Y, V : Y \triangleright_V \iota(A); \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash V\ \mathtt{id}_p\ \tau[\uparrow^t V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)\ (\uparrow^{\sigma'} \llbracket F \rrbracket_F) : \bot}\;(@)}{\Gamma \vdash \lambda \sigma \tau V.V\ \mathtt{id}_p\ \tau[\uparrow^t V](\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)\ (\uparrow^{\sigma'} \llbracket F \rrbracket_F) : \llbracket \Gamma \rrbracket_\Gamma \triangleright_F \iota(A)}\;(\lambda)$$

where:

- $n = |\tau|$, $k = n - i$, $p = n + |\tau'|$, $\sigma' = \sigma \circ \delta_{[i,p]}^{+k}$
- Π_F is the following proof, obtained by Lemma 6.49:

$$\dfrac{; \vdash F : (\llbracket \Gamma \rrbracket_\Gamma, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A) \qquad \overline{\sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma}\;(\text{Ax})}{; \sigma : Y <: \llbracket \Gamma' \rrbracket_\Gamma \vdash (\uparrow^{\sigma'} F) : (Y, \iota(A), \llbracket \Gamma' \rrbracket_\Gamma) \triangleright_F \iota(A)}$$

- Π_τ is the following proof:

$$\dfrac{\dfrac{}{\tau : Y \vdash \tau : Y}\,(\mathrm{Ax}) \quad \dfrac{\dfrac{\dfrac{}{V : Y \triangleright_V \iota(A) \vdash V : Y \triangleright_V \iota(A)}\,(\mathrm{Ax})}{V : Y \triangleright_V \iota(A) \vdash\uparrow^t V : Y \triangleright_t \iota(A)}\,(\uparrow)}{V : Y \triangleright_V \iota(A) \vdash [V] : Y \triangleright_\tau \iota(A)}\,(\tau_t)}{\dfrac{\tau : Y, V : Y \triangleright_V \iota(A) \vdash \tau[\uparrow^t V] : Y, \iota(A) \qquad \mathbf{\Pi}_{\tau'}}{\tau : Y, V : Y' \triangleright_V \iota(A); \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \tau[\uparrow^t V]\llbracket\tau'\rrbracket_\tau : (Y, \iota(A), \llbracket\Gamma'\rrbracket^{\sigma[x:=n]})}\,(\tau\tau')}\,(\tau\tau')$$

- $\Pi_{\tau'}$ is the following proof, obtained from the induction hypothesis for τ' and Lemma 6.49:

$$\dfrac{\vdash \llbracket\tau'\rrbracket_\tau : \llbracket\Gamma\rrbracket_\Gamma, \iota(A) \triangleright_\tau \llbracket\Gamma'\rrbracket_\Gamma \quad \dfrac{}{\sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})}{; \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash\uparrow^{\sigma'} \llbracket\tau'\rrbracket_\tau : Y, \iota(A) \triangleright_\tau \llbracket\Gamma'\rrbracket_\Gamma}$$

5. Terms

- **Case** $\llbracket V\rrbracket_t$. This case is similar to the case $\llbracket v\rrbracket_V$.

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

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

Hence we have by induction a proof of $; \vdash \llbracket c\rrbracket_c : \llbracket\Gamma, x_i : A^{\perp\!\!\!\perp}\rrbracket_\Gamma \triangleright_c \bot$ and we can derive:

$$\dfrac{\dfrac{\dfrac{; \vdash \llbracket c\rrbracket_c : \llbracket\Gamma, x_i : A^{\perp\!\!\!\perp}\rrbracket_\Gamma \triangleright_c \bot \qquad \mathbf{\Pi}_\sigma}{\tau : Y; \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \llbracket c\rrbracket_c\ \sigma^+_{|\tau|} : (Y, \iota(A)^{\perp\!\!\!\perp}) \to \bot}\,(\forall_E) \qquad \mathbf{\Pi}_\tau}{\tau : Y, E : Y \triangleright_E \iota(A); \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \llbracket c\rrbracket_c\ \sigma^+_{|\tau|}\ \tau[E] : \bot}\,(@)}{; \vdash \lambda\sigma\tau E.\llbracket c\rrbracket_c\ \sigma^+_{|\tau|}\ \tau[E] : \llbracket\Gamma\rrbracket_\Gamma \triangleright_t \iota(A)}\,(\lambda)$$

where

- Π_σ is the following derivation, obtained by Lemma 6.48 (since $|\tau|$ matches $|Y|$):

$$\dfrac{\dfrac{}{\sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma}\,(<:_{\mathsf{ax}})}{\sigma : Y <: \llbracket\Gamma\rrbracket_\Gamma \vdash \sigma^+_{|\tau|} : (Y, \iota(A)^{\perp\!\!\!\perp}) <: \llbracket\Gamma, x_i : \iota(A)^{\perp\!\!\!\perp}\rrbracket_\Gamma}$$

- Π_E is also obtained by Lemma 6.48:

$$\dfrac{\dfrac{}{\tau : Y, E : Y \triangleright_E \iota(A); \vdash \tau[E] : Y, \iota(A)^{\perp\!\!\!\perp}}\,(\mathrm{Ax}) \quad \dfrac{}{E : Y \triangleright_E \iota(A); \vdash E : Y \triangleright_E \iota(A)}\,(\mathrm{Ax})}{\tau : Y, E : Y \triangleright_E \iota(A); \vdash \tau[E] : Y, \iota(A)^{\perp\!\!\!\perp}}$$

6. Contexts

- **Case** $\llbracket E\rrbracket_e$. This case is similar to the case $\llbracket v\rrbracket_V$.

- **Case** $\llbracket\tilde{\mu}x_i.c\rrbracket_e$. This case is similar to the case $\llbracket\mu\alpha_i.c\rrbracket_t$.

7. Commands

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

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

thus we get by induction two proofs of $; \vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \triangleright_t \iota(A)$ and $; \vdash \llbracket e \rrbracket_c : \llbracket \Gamma \rrbracket_\Gamma \triangleright_e \iota(A)$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{; \vdash \llbracket e \rrbracket_e : \llbracket \Gamma \rrbracket_\Gamma \triangleright_e \iota(A)}{\tau : Y; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \llbracket e \rrbracket_e\ \sigma : Y \to Y \triangleright_t \iota(A) \to \bot}\,(\forall_E) \quad \mathbf{\Pi}_\sigma}{\tau : Y; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \llbracket e \rrbracket_e\ \sigma\ \tau : Y \triangleright_t \iota(A) \to \bot}\,(@) \quad \dfrac{}{\tau : Y; \vdash \tau : Y}\,(\text{Ax})}{\tau : Y; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \llbracket e \rrbracket_e\ \sigma\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t) : \bot}\,(@) \quad \mathbf{\Pi}_t}{; \vdash \lambda \sigma \tau . \llbracket e \rrbracket_e\ \sigma\ \tau\ (\uparrow^\sigma \llbracket t \rrbracket_t) : \llbracket \Gamma \rrbracket_\Gamma \triangleright_c \bot}\,(\lambda)$$

where:

- Π_σ is the axiom rule: $\dfrac{}{\sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma}\,(<:_{\text{ax}})$
- Π_t is obtained using Lemma 6.45:

$$\frac{; \vdash \llbracket t \rrbracket_t : \llbracket \Gamma \rrbracket_\Gamma \triangleright_t \iota(A) \quad \dfrac{}{; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma}\,(<:_{\text{ax}})}{; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \uparrow^\sigma \llbracket t \rrbracket_t : Y \triangleright_t \iota(A)}$$

8. Closures

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

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

where n matches $|\Gamma|$. We thus get by induction two proofs $; \vdash \llbracket \tau' \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma$ and $\vdash \llbracket c \rrbracket_c : \llbracket \Gamma, \Gamma' \rrbracket_\Gamma \triangleright_c \bot$. We can derive:

$$\dfrac{\dfrac{\dfrac{\vdash \llbracket c \rrbracket_c : \llbracket \Gamma, \Gamma' \rrbracket_\Gamma \triangleright_c \bot \qquad \Pi_\sigma}{; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \llbracket c \rrbracket_c\ \sigma' : (Y, \llbracket \Gamma' \rrbracket_\Gamma) \to \bot}\,(\forall_E) \quad \dfrac{\dfrac{}{\tau : Y; \vdash \tau : Y}\,(\text{Ax}) \qquad \mathbf{\Pi}_\tau}{\tau : Y; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \tau(\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau) : Y \llbracket \Gamma' \rrbracket_\Gamma}\,(\tau\tau')}{\tau : Y; \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \llbracket c \rrbracket_c\ \sigma'\ \tau'(\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau : \bot}\,(@)}{; \vdash \lambda \sigma \tau . \llbracket c \rrbracket_c\ \sigma'\ \tau'(\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau)}\,(\lambda)$$

where $k = |\tau'| - n$, $p = n + |\tau|$, $\sigma' = \sigma \circ \delta_{[n,p]}^{+k}$ and:

- Π_σ is a proof of $\sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma \vdash \sigma' : (Y, \llbracket \Gamma' \rrbracket_\Gamma) <: \llbracket \Gamma, \Gamma' \rrbracket_\Gamma$ obtained by Lemma 6.49;
- $\Pi_{\tau'}$ is the following proof also obtained by Lemma 6.49:

$$\frac{; \vdash \llbracket \tau' \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma \quad \dfrac{}{\vdash \sigma : Y <: \llbracket \Gamma \rrbracket_\Gamma}\,(<:_{\text{ax}})}{\vdash (\uparrow^{\sigma'} \llbracket \tau' \rrbracket_\tau) : Y \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma}$$

9. Stores

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

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

By induction, we obtain two proofs of $\vdash \llbracket \tau \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma$ and $\vdash \llbracket t \rrbracket_t : \llbracket \Gamma, \Gamma' \rrbracket_\Gamma \triangleright_t \iota(A)$. We can thus derive:

$$\dfrac{\vdash \llbracket \tau \rrbracket_\tau : \llbracket \Gamma \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma \quad \dfrac{\vdash \llbracket t \rrbracket_t : \llbracket \Gamma, \Gamma' \rrbracket_\Gamma \triangleright_t \iota(A)}{\vdash [\llbracket t \rrbracket_t] : \llbracket \Gamma, \Gamma' \rrbracket_\Gamma \triangleright_\tau \iota(A)}\,(\tau_t)}{\vdash \llbracket \tau \rrbracket_\tau [\llbracket t \rrbracket_t] : \llbracket \Gamma \rrbracket_\Gamma \triangleright_\tau \llbracket \Gamma' \rrbracket_\Gamma, \iota(A)}\,(\tau\tau')$$

□

6.5 Conclusion and perspectives

6.5.1 Conclusion

In this chapter, we presented a system of simple types for a call-by-need calculus with control. We proved that this type system is safe, in the sense that it satisfies the subject reduction property (Theorem 6.2) and the (weak) normalization property (Theorem 6.22). We proved the normalization by means of realizability-inspired interpretation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus. Incidentally, this opens the doors to the computational analysis (in the spirit of Krivine classical realizability) of classical proofs using control, laziness and shared memory.

Besides, we introduced system F_Υ as a type system for the target of a continuation-and-store-passing style translation for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, and we proved that the translation was well-typed (Theorem 6.33). Furthermore, we also refined our presentation to define both source and target languages with explicit De Bruijn levels, making them both more compatible with an implementation.

Last, we believe that the principles guiding the typing of the translation emphasized its computational content, whose three main ingredients are the following:

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

The latter is particularly highlighted in the translation with De Bruijn levels, where levels need to be shifted when extending the store and coercions give a computational content to the subtyping relation (*i.e.* to store extension).

6.5.2 About stores and forcing

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

$$[\![A \to B]\!] \triangleq S \times A \to S \times B$$

In particular, the result of a function may depend on the current state. If one observes precisely our realizability interpretation, it is very similar to our definition of truth and falsity values: for a type A, its interpretation is roughly of the shape $A \times \tau$. It is folklore that the state monad can be categorically interpreted by means of presheaves construction [138, 116]. Interestingly, Kripke models are a particular case of presheaves semantics [123]. Cohen forcing construction is also interpreted in terms of presheaves [111], and this interpretation scales to type theory [82, 81]. Therefore, the state monad and the forcing translation were already known to be connected. Last but not least, the analysis of Cohen forcing in the framework of Krivine classical realizability [98, 120] relies on an extension of Krivine abstract machine with a cell (which contains the forcing condition). In short, our typed store-passing style translation is just another observation of the connection between forcing translations and explicit stores as a side-effect.

6.5.3 Extension to 2$^{\text{nd}}$-order type systems

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

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

We can then define the following rules to introduce the universal quantification:

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

Observe that these rules need to be restricted at the level of strong values, just as they are restricted to values in the case of call-by-value (see Section 4.5.4). As for the left rules, they can be defined at any levels, let say the more general e:

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

where n is any natural number and B any formula. The usual (call-by-value) interpretation of the quantification is defined as an intersection over all the possible instantiations of the variables within the model. First-order variables are to be instantiated by integers, while second-order variables are to be instantiated by sets of terms at the lowest level, *i.e.* closed strong-values in store (which we write $\mathcal{V}_0$):

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

It is then routine to check that the typing rules are adequate with the realizability interpretation.

6.5.4 Related work & further work

In a recent paper, Kesner uses an intersection type system to characterize normalizing by-need terms [86]. Even though her calculus is not classical, it might be interesting to adapt her approach to our framework. Specifically, we have the intuition that intersection types could be an alternative to our subtyping relation in the target language of the CPS.

As for call-by need with control, recent work by Pédrot and Saurin [134] relates (classical) call-by-need with linear head-reduction from a computational point of view. If they do not provide any type system or normalization results, they connect their framework with a variant of the $\overline{\lambda}_{lv}$-calculus (in natural deduction style). Our techniques should then be adaptable to their framework in order to equip their calculi with type systems and prove similar results.

This chapter naturally raises the question of studying the system F_Υ that we used as target language of our translation. In particular, it might be interesting to understand the logical strength of such a system. It seems to be stronger than systems F or F$_{<:}$ in that is allows a restricted form of dependent types: the second-order quantification range over vectors of arbitrary size. It is probably weaker than a higher order calculus with unrestricted dependencies in types, like the calculus of constructions (which is logically as strong as F$_\omega$). Yet, it might also be the case that a clever analysis of the translation could lead to a bound on the size of the store extension at each step. This would offer a way to remove this dependency and to embed the target language into system F.