



**HAL**  
open science

# A continuation-passing-style interpretation of simply-typed call-by-need $\lambda$ -calculus with control within System F

Hugo Herbelin, Étienne Miquey

► **To cite this version:**

Hugo Herbelin, Étienne Miquey. A continuation-passing-style interpretation of simply-typed call-by-need  $\lambda$ -calculus with control within System F. 2016. hal-01302696v1

**HAL Id: hal-01302696**

**<https://inria.hal.science/hal-01302696v1>**

Preprint submitted on 14 Apr 2016 (v1), last revised 25 May 2016 (v2)

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# A continuation-passing-style interpretation of simply-typed call-by-need $\lambda$ -calculus with control within System F

Hugo Herbelin

Inria - IRIF - PPS -  $\pi.r^2$  - University Paris Diderot

Étienne Miquey

IRIF - PPS -  $\pi.r^2$  - University Paris Diderot  
University of Montevideo

Ariola *et al* defined a call-by-need  $\lambda$ -calculus with control, together with a sequent calculus presentation of it, and a mechanically generated continuation-passing-style transformation simulating the reduction. We present here a simply-typed version of this calculus and shows that it maps to System F through the continuation-passing-style transformation. This implies in particular the normalization of this simply-typed call-by-need calculus with control. Incidentally, we treat bound variables for the continuation-passing-style transformation in a precise way using indices rather than up to  $\alpha$ -conversion, what makes it directly implementable.

## Introduction

**Call-by-name, call-by-value and call-by-need evaluation strategies** The call-by-name and call-by-value evaluation strategies are two basic strategies for evaluating  $\lambda$ -calculus. The call-by-name evaluation strategy passes arguments to functions without evaluating them, postponing their evaluation to each place the argument is needed, re-evaluating it several times if needed several times.

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, but, if ever the argument is not needed, it is evaluated uselessly.

The call-by-need is a third evaluation strategy of  $\lambda$ -calculus which evaluates arguments of functions only when needed, and, when needed, shares their evaluation across all places where the argument is needed. Call-by-need evaluation is at the heart of a functional programming language such as Haskell. Intentionally, it behaves like the call-by-value evaluation strategy in the sense that all places where a same argument is used share the same value. Observationally, it however 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 call-by-name evaluation. In a setting with non-terminating computations, it is not observationally equivalent to call-by-value evaluation since if the evaluation of a useless argument loops in call-by-value evaluation, the whole computation loops, which is not the case of call-by-name and call-by-need evaluation.

**Call-by-name, call-by-value and call-by-need calculi** The call-by-name, call-by-value and call-by-need evaluation strategies can be turned into equational theories. This has been done by Plotkin [14] who introduced call-by-name and call-by-value continuation-passing-style semantics. For call-by-name, the corresponding induced equational theory is actually Church’s original theory of  $\lambda$ -calculus based on the operational rule  $\beta$  and the extensional rule  $\eta$ .

For call-by-value, Plotkin showed that the induced equational theory includes the key operational rule  $\beta_V$  and the extensional rule  $\eta_V$ . The induced equational theory was further completed implicitly

by Moggi [11] with the convenient introduction of a native `let` operator<sup>1</sup>. It was then explicitly shown complete by Sabry and Felleisen [15].

For the call-by-need evaluation strategy, a proper equational theory reflecting the strategy into a semantics was proposed independently by Ariola-Felleisen [1] and Maraist-Odersky-Wadler [10] which emphasize the intentional behavior call-by-need, though not complete enough, in the sense that it cannot show in general that call-by-need and call-by-name observationally coincide for  $\lambda$ -calculus.

For call-by-need, a continuation-passing-style semantics was proposed by Okasaki-Lee-Tarditi [12] but it happens that this semantics does not ensure normalization of simply-typed call-by-need evaluation, thus failing to ensure a property which however holds in the call-by-name and call-by-value cases.

Building on top of the duality between programs and their evaluation contexts [6], and the duality between `let` constructs (which bind programs) and a control operator such as Parigot's  $\mu$  (which bind evaluation contexts), the first author proposed the core of a call-by-need reduction semantics supporting control operators [8]. This semantics was studied in Ariola *et al* [4], then [2], eventually providing a continuation-passing-style semantics. It is this semantics that we study in this paper.

Continuation-passing-style semantics de facto gives a semantics to control operators, i.e. to operators such as Scheme's `callcc`, Felleisen's  $\mathcal{C}$ ,  $\mathcal{H}$ , or  $\mathcal{A}$  operators [7], Parigot's  $\mu$  and  $[\ ]$  operators [13], Crolard's `catch` and `throw` operators [5]. In particular, even though call-by-name and call-by-need are observationally equivalent on pure  $\lambda$ -calculus, their different intentional behavior induces different continuation-passing-style semantics, and this reflects that they behave observationally differently when control operators are considered.

**Continuation-passing-style for simply-typed call-by-need calculus with control** We shall concentrate on typing the continuation-passing-style transformation presented in [2]. Since evaluation of terms is shared, this continuation-passing-style is actually combined with an environment-passing-style transformation. Moreover, the environment can grow, so the translation also includes a Kripke-style forcing to address the extensibility of the store.

We shall focus on one of the calculi presented in [2], namely  $\bar{\lambda}_{[V\tau^*]}$ , even though the treatment could be done for the simpler calculus  $\bar{\lambda}_V$  as well. The calculi  $\bar{\lambda}_{[V\tau^*]}$  is a sequent calculus targeted to call-by-need. We remind its syntax in Section 1 before equipping it with a system of simple types in Section 2. The core of the paper is in typing the continuation-passing-style translation, what is done in Section 3.

## 1 Syntax

We remind the syntax of  $\bar{\lambda}_{[V\tau^]}$  from [2], introducing a split of the notion of values from [2] into two categories: strong values ( $v$ ) and weak values ( $V$ ). The strong values correspond to values properly

---

<sup>1</sup>In Plotkin, `let  $x = t$  in  $u$`  is simulated by  $(\lambda x.u)t$ , but the latter fails to satisfy a Gentzen-style principle of “purity of methods” as it requires to know the constructor  $\lambda$  and destructor application of an arrow type for expressing something which is just a cut rule and has no reason to know about the arrow type. This is the same kind of purity of methods as in natural deduction compared to Frege-Hilbert systems: the latter uses the connective  $\rightarrow$  to internalize derivability  $\vdash$  leading to require  $\rightarrow$  even when talking about the properties of say,  $\wedge$ . This is the same kind of purity of methods as in Parigot's classical natural deduction and  $\lambda\mu$ -calculus compared to say Prawitz's extension of natural deduction with Reduction ad absurdum: the latter uses the connective  $\perp$  to internalize judgments with “no conclusion” and uses the connective  $\neg$  to internalize the type of “evaluation contexts” (i.e. co-terms). See Curien-Herbelin [6] for a calculus emphasizing the proof-as-program correspondence between “no conclusion” judgments and states of an abstract machine, between right-focused judgments and programs, and between left-focused judgments and evaluation contexts. Krivine [9], followed by Ariola *et al* [3] have a convenient notation  $\perp\perp$  to characterize such “no conclusion” judgments.

speaking. The weak values includes the variables which force the evaluation of terms to which they refer into shared strong value. Their evaluation may require capturing a continuation.

For binders binding terms, we use De Bruijn levels, i.e. names of the form  $x_i$  where  $x$  is a fixed name serving just the purpose of looking like a name and where the relevant information is the number  $i$  which counts how many term binders have been already traversed from the root of the term. For binders binding evaluation contexts, we similarly use De Bruijn levels, but with variables of the form  $\alpha_i$ , where, again,  $\alpha$  is a fixed name indicating that the variable is binding evaluation contexts and not terms. Note that binders, substituting a term  $t$  within an other term requires in general to renumbered the bound variables of  $t$ , shifting them by the number of binders traversed by  $t$  before reaching the positions where it is substituted. The syntax of the language is given by:

Strong values	$v ::= \lambda x_i.t$
Weak values	$V ::= v \mid x_i$
Terms	$t ::= V \mid \mu \alpha_i.c$
Forcing contexts	$F ::= \alpha \mid t \cdot E$
Catchable contexts	$E ::= F \mid \alpha_i \mid \tilde{\mu}[x_i].\langle x_i \parallel F \rangle \tau$
Evaluation contexts	$e ::= E \mid \tilde{\mu}x_i.c$
Commands	$c ::= \langle t \parallel e \rangle$
Substitutions	$\tau ::= \varepsilon \mid \tau[x_i := t]$

Types and typing contexts are defined by:

$$A, B ::= X \mid A \rightarrow B$$

$$\Gamma ::= \varepsilon \mid \Gamma, x : A$$

$$\Delta ::= \varepsilon \mid \Delta, \alpha : A$$

The reduction rules are given on Figure 1.

$\langle t \parallel \tilde{\mu}x.c \rangle \tau$	$\rightarrow$	$c\tau[x := t]$
$\langle \mu \alpha.c \parallel E \rangle \tau$	$\rightarrow$	$(c[E/\alpha])\tau$
$\langle x \parallel F \rangle \tau[x := t] \tau'$	$\rightarrow$	$\langle t \parallel \tilde{\mu}[x].\langle x \parallel F \rangle \tau' \rangle \tau$
$\langle V \parallel \tilde{\mu}[x].\langle x \parallel F \rangle \tau' \rangle \tau$	$\rightarrow$	$\langle V \parallel F \rangle \tau[x := V] \tau'$
$\langle \lambda x.t \parallel u \cdot E \rangle \tau$	$\rightarrow$	$\langle u \parallel \tilde{\mu}x.\langle t \parallel E \rangle \rangle \tau$

Figure 1: Reduction rules

## 2 Typing rules

We have eight kinds of sequents, one for typing each of the eight syntactic categories. We write them with an annotation on the  $\vdash$  sign: one of the letters  $v, V, t, F, E, e, c, \tau$ . 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 with the type written on the left; sequents typing commands are black box neither asserting nor expecting a type; sequents typing substitutions are instantiating a typing context with the substitution and its type written on the right. Otherwise said, we have the following eight kinds of sequents:

$$\begin{array}{c}
\frac{\Gamma, x_n : A \vdash_t \Delta; t : B \quad |\Gamma| = n}{\Gamma \vdash_v \Delta; \lambda x_n. t : A \rightarrow B} \\
\\
\frac{\Gamma(i) = x_i : A}{\Gamma \vdash_v \Delta; x_i : A} \quad \frac{\Gamma \vdash_v \Delta; v : A}{\Gamma \vdash_V \Delta; v : A} \\
\\
\frac{\Gamma \vdash_V \Delta; V : A}{\Gamma \vdash_t \Delta; V : A} \quad \frac{c : (\Gamma \vdash_c \Delta, \alpha_n : A) \quad |\Delta| = n}{\Gamma \vdash_t \Delta; \mu \alpha_n. c : A} \\
\\
\frac{}{\Gamma; \alpha : A \vdash_F \Delta} \quad \frac{\Gamma; \vdash_t \Delta; t : A \quad \Gamma; E : B \vdash_E \Delta}{\Gamma; t \cdot E \vdash_F \Delta} \\
\\
\frac{\Delta(i) = \alpha_i : A}{\Gamma; \alpha_i : A \vdash_E \Delta} \quad \frac{\Gamma; F : A \vdash_F \Delta \quad \Gamma, x_n : A, \Gamma'; F : A \vdash_F \Delta \quad \Gamma \vdash \Delta; \tau : \Gamma' \quad |\Gamma| = n}{\Gamma; \tilde{\mu}[x_n]. \langle x_n \| F \rangle \tau : A \vdash_E \Delta} \\
\\
\frac{\Gamma; E : A \vdash_E \Delta}{\Gamma; E : A \vdash_e \Delta} \quad \frac{c : (\Gamma, x_n : A \vdash_t \Delta) \quad |\Gamma| = n}{\Gamma; \tilde{\mu} x_n. c : A \vdash_e \Delta} \\
\\
\frac{\Gamma \vdash_t \Delta; t : A \quad \Gamma; e : A \vdash_e \Delta}{\langle t \| e \rangle : (\Gamma \vdash_c \Delta)} \quad \frac{c : (\Gamma, \Gamma' \vdash_c \Delta) \quad \Gamma \vdash_\tau \Delta; \tau : \Gamma'}{c\tau : (\Gamma \vdash_\tau \Delta)} \\
\\
\frac{}{\Gamma \vdash_\tau \Delta; \varepsilon : \varepsilon} \quad \frac{\Gamma \vdash_\tau \Delta; \tau : \Gamma' \quad \Gamma, \Gamma' \vdash_t \Delta; t : A}{\Gamma \vdash_\tau \Delta; \tau[x := t] : \Gamma', x : A}
\end{array}$$

Figure 2: Typing rules

$$\begin{array}{l}
\Gamma \vdash_v \Delta; v : A \\
\Gamma \vdash_V \Delta; V : A \\
\Gamma \vdash_t \Delta; t : A \\
\Gamma; F : A \vdash_F \Delta \\
\Gamma; E : A \vdash_E \Delta \\
\Gamma; e : A \vdash_e \Delta \\
c : (\Gamma \vdash_c \Delta) \\
\Gamma \vdash_\tau \Delta; \tau : \Gamma
\end{array}$$

The typing rules are given on Figure 2 where  $|\Gamma|$  denotes the length of  $\Gamma$ , and  $\Gamma(i)$  for  $i < |\Gamma|$  denotes the  $i^{\text{th}}$  in  $\Gamma$ , and similarly for  $|\Delta|$  and  $\Delta(i)$ .

<i>Syntax</i>	
Types	$T, U ::= X \mid T \rightarrow U \mid \forall X. T$
Terms	$t, u ::= x \mid \lambda x. t \mid tu \mid \Lambda X. t \mid tT$
Typing contexts	$\Gamma ::= \varepsilon \mid \Gamma, x : T \mid \Gamma, X$
where $X$ ranges over countably many type variables	
 <i>Typing rules</i>	
$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$	
$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x. t : U \rightarrow T} \quad \frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash_V u : U}{\Gamma \vdash tu : T}$	
$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X. t : \forall X. T} \quad \frac{\Gamma \vdash t : \forall X. T}{\Gamma \vdash tU : T[X := U]}$	

Figure 3: Syntax and typing rules of System F

### 3 A typed cps-translation

We shall rephrase the continuation-passing-style transformation of  $\bar{\lambda}_{[lv\tau^*]}$  from [2], typing it and using indices for variables.

We shall first give a translation of typing sequents for  $\bar{\lambda}_{[lv\tau^*]}$  into typing sequents of System F, whose syntax and typing rules “à la Curry” are reminded on Figure 3. In System F, one can define a unit type  $\top \triangleq \forall X. X \rightarrow X$  with canonical inhabitant  $() \triangleq \Lambda X. \lambda x_i. x_i$  for appropriate  $i$ . Similarly, one can define a  $n$ -ary conjunction  $T_1 \wedge \dots \wedge T_n \triangleq \forall X. (T_1 \rightarrow \dots \rightarrow T_n \rightarrow X) \rightarrow X$  with constructor  $\langle t_1, \dots, t_n \rangle \triangleq \Lambda X. \lambda x_i. x_i t_1 \dots t_n$  for appropriate  $i$ . Projections can then be defined as well. Finally, one can define  $\perp \triangleq \forall X. X$ .

For the purpose of the translation on types, we omit the terms, values, contexts, etc. and concentrate only the types. In the following,  $\vec{T}$  is a sequence of System F types of length the number of declarations in  $\Gamma$  plus one.

The transformation is actually not only a continuation-passing-style translation. Because of sharing of the evaluation of arguments, the environment associating terms to variables behaves like a store which is passed around. Passing the store amounts to combine the continuation-passing-style translation with an environment-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.

Let us explain step by step the rationale guiding the definition of the translation, and, in a first approximation, let us look only at the continuation-passing-style part of the translation of a  $\bar{\lambda}_{[lv\tau^*]}$  sequent. Then, the translation of a sequent such as  $\Gamma \vdash_t \Delta; A$  is a System F sequent  $\dashv\vdash\vdash\Delta, \dashv\vdash\vdash\Gamma \vdash \dashv\vdash\vdash A$ .

There are respectively 3 negations over  $\Delta$  and 4 over  $\Gamma$  and  $A$  because, as shown in [2] and as emphasized by the 6 nested syntactic categories used to define  $\bar{\lambda}_{[\nu\tau\star]}$ , there are 6 levels of control in call-by-need, leading to 6 mutually defined levels of interpretation:

- $\llbracket A \rrbracket_v$  for strong values: a strong value of type  $A \rightarrow B$  is interpreted as a term of type  $\llbracket A \rightarrow B \rrbracket_v \triangleq \llbracket A \rrbracket_t \rightarrow \llbracket B \rrbracket_E \rightarrow \perp$ , i.e., informally,  $\neg\neg\neg\neg A \rightarrow \neg\neg\neg\neg B$ .
- $\llbracket A \rrbracket_F$  for forcing contexts: a forcing context is expecting to take control over a strong value, returning to the toplevel or passing arguments to it, so  $\llbracket A \rrbracket_F \triangleq \llbracket A \rrbracket_v \rightarrow \perp$ , i.e., informally,  $\neg A$ .
- $\llbracket A \rrbracket_V$  for weak values: a weak value is expecting to take control over a forcing context, possibly duplicating or erasing it in the case of a variable bound to a  $\mu\alpha.c$  construct in the shared environment, so  $\llbracket A \rrbracket_V \triangleq \llbracket A \rrbracket_F \rightarrow \perp$ , i.e., informally  $\neg\neg A$ .
- $\llbracket A \rrbracket_E$  for catchable evaluation contexts: a catchable evaluation context is expecting to take control over a weak value, possibly duplicating or erasing it with the  $\tilde{\mu}[x].\langle x \parallel F \rangle \tau$  construct, so  $\llbracket A \rrbracket_E \triangleq \llbracket A \rrbracket_V \rightarrow \perp$ , i.e., informally  $\neg\neg\neg A$ .
- $\llbracket A \rrbracket_t$  for terms: a term is expecting to take control over a catchable evaluation context, possibly duplicating or erasing it with the  $\mu\alpha.c$  construct, so  $\llbracket A \rrbracket_t \triangleq \llbracket A \rrbracket_E \rightarrow \perp$ , i.e., informally  $\neg\neg\neg\neg A$ .
- $\llbracket A \rrbracket_e$  for general evaluation contexts: a general evaluation context is expecting to take control over a term, possibly duplicating or erasing it with the  $\tilde{\mu}x.c$  construct, so  $\llbracket A \rrbracket_e \triangleq \llbracket A \rrbracket_t \rightarrow \perp$ , i.e., informally  $\neg\neg\neg\neg\neg A$ .

In particular, when translating a sequent such as  $\Gamma \vdash_t \Delta; A$ , the context  $\Delta$  is interpreted as a context of catchable evaluation contexts while  $\Gamma$  is interpreted at the level of terms since it carries the store whose components are terms to be evaluated in a shared way. Otherwise said, in the particular case of  $\Gamma \vdash_t \Delta; A$  the translation is  $\llbracket \Delta \rrbracket_E, \llbracket \Gamma \rrbracket_t \vdash \llbracket A \rrbracket_t$ , and similarly for other levels, e.g.,  $\Gamma; A \vdash_e \Delta$  translates to  $\llbracket \Delta \rrbracket_E, \llbracket \Gamma \rrbracket_t \vdash \llbracket A \rrbracket_e$ .

The continuation-passing-style part being settled, the environment-passing-style part should be considered. In particular, the translation of  $\Gamma \vdash_t \Delta; A$  is not anymore a sequent  $\llbracket \Delta \rrbracket_E, \llbracket \Gamma \rrbracket_t \vdash \llbracket A \rrbracket_t$  but instead a sequent roughly of the form  $\llbracket \Delta \rrbracket_E \vdash \llbracket \Gamma \rrbracket_t \rightarrow \llbracket A \rrbracket_t$ , with actually  $\Gamma$  being passed around not only at the top level of  $\llbracket A \rrbracket_t$  but also every time a negation is used. Additionally, abstraction over  $\Gamma$  should be passed around over all types in  $\Delta$ . And, moreover, the translation of each type in  $\Gamma$  should itself be abstracted over the store at each use of a negation, so for instance, at this step of the explanation, the translation of  $X_1, X_2 \vdash_v Y; X$  is  $\llbracket X_1 \rrbracket_t, \llbracket X_2 \rrbracket_t, \llbracket Y \rrbracket_E \vdash \llbracket X \rrbracket_v$ , where:

- $\llbracket X_1 \rrbracket_t$  is  $\neg\neg\neg\neg X_1$ ,
- $\llbracket X_2 \rrbracket_t$ , dependent on  $\llbracket X_1 \rrbracket_t$  at each level of negation is  $\llbracket X_1 \rrbracket_t \rightarrow \neg(\llbracket X_1 \rrbracket_t \rightarrow \neg(\llbracket X_1 \rrbracket_t \rightarrow \neg(\llbracket X_1 \rrbracket_t \rightarrow \neg X_2)))$ ,
- $\llbracket Y \rrbracket_E$ , similarly dependent on  $\llbracket X_1 \rrbracket_t$  and  $\llbracket X_2 \rrbracket_t$  at each level is  $(\llbracket X_1 \rrbracket_t \rightarrow \llbracket X_2 \rrbracket_t \rightarrow \neg(\llbracket X_1 \rrbracket_t \rightarrow \llbracket X_2 \rrbracket_t \rightarrow \neg(\llbracket X_1 \rrbracket_t \rightarrow \llbracket X_2 \rrbracket_t \rightarrow \neg Y)))$ ,
- $\llbracket X \rrbracket_v$ , also dependent on  $\llbracket X_1 \rrbracket_t$  and  $\llbracket X_2 \rrbracket_t$  at each level is  $\llbracket X_1 \rrbracket_t \rightarrow \llbracket X_2 \rrbracket_t \rightarrow \neg(\llbracket X_1 \rrbracket_t \rightarrow \llbracket X_2 \rrbracket_t \rightarrow \neg X)$ .

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 of the store. Schematically, this is obtained by typing the store with types  $T_0, A_1, T_1, \dots, A_n, T_n$  for arbitrary  $T_0, \dots, T_n$ , whenever the typing context is  $A_1, \dots, A_n$ , and to have each of these  $T_i$  extensible. The extensibility is obtained by quantification over all possible extensions of  $T_i$  at each level of the negation.

The resulting translation on judgments and types is given in Figure 4 and Figure 5 where:

$$\begin{array}{l}
\llbracket \Gamma; e : A \vdash_e \Delta \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket e \rrbracket_e^{\vec{T}} : \Gamma \triangleright_e^{\vec{T}} A \\
\llbracket \Gamma \vdash_t \Delta; t : A \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket t \rrbracket_t^{\vec{T}} : \Gamma \triangleright_g^{\vec{T}} A \\
\llbracket \Gamma; E : A \vdash_E \Delta \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket E \rrbracket_E^{\vec{T}} : \Gamma \triangleright_E^{\vec{T}} A \\
\llbracket \Gamma \vdash_v \Delta; V : A \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket V \rrbracket_V^{\vec{T}} : \Gamma \triangleright_V^{\vec{T}} A \\
\llbracket \Gamma; F : A \vdash_F \Delta \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket F \rrbracket_F^{\vec{T}} : \Gamma \triangleright_F^{\vec{T}} A \\
\llbracket \Gamma \vdash_v \Delta; v : A \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket v \rrbracket_v^{\vec{T}} : \Gamma \triangleright_v^{\vec{T}} A \\
\llbracket c : (\Gamma \vdash_c \Delta) \rrbracket^{\vec{T}} \triangleq \Gamma \triangleright^{\vec{T}} \Delta \vdash \llbracket c \rrbracket_c^{\vec{T}} : (\Gamma \triangleright_{\geq \vec{T}} \perp)
\end{array}$$

Figure 4: Translation of judgments

$$\begin{array}{l}
\Gamma \triangleright_e^{\vec{T}} A \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_t^{\vec{X}} A \rightarrow \perp) \\
\Gamma \triangleright_t^{\vec{T}} A \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_E^{\vec{X}} A \rightarrow \perp) \\
\Gamma \triangleright_E^{\vec{T}} A \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_V^{\vec{X}} A \rightarrow \perp) \\
\Gamma \triangleright_V^{\vec{T}} A \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_F^{\vec{X}} A \rightarrow \perp) \\
\Gamma \triangleright_F^{\vec{T}} A \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_v^{\vec{X}} A \rightarrow \perp) \\
\Gamma \triangleright_v^{\vec{T}} A \rightarrow B \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_t^{\vec{X}} A \rightarrow \Gamma \triangleright_E^{\vec{X}} B \rightarrow \perp) \\
\Gamma \triangleright_v^{\vec{T}} Y \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} Y \\
\varepsilon \triangleright^{Y \geq U} B(Y) \triangleq \forall Y \geq U. Y \rightarrow B(Y) \\
\Gamma, x : A \triangleright_{\geq \vec{T}}^{\vec{X}} B(\vec{X}, Y) \triangleq \Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} (\Gamma \triangleright_t^{\vec{X}} A \rightarrow \forall Y \geq U. Y \rightarrow B(\vec{X}, Y)) \\
\Gamma \triangleright^{\vec{T}} \varepsilon \triangleq \varepsilon \\
\Gamma \triangleright^{\vec{T}} (\Delta, \alpha : A) \triangleq (\Gamma \triangleright^{\vec{T}} \Delta), (\Gamma \triangleright_E^{\vec{T}} A)
\end{array}$$

Figure 5: Translation of types

- $\Gamma \triangleright^{\vec{T}} \Delta$  denotes the translation of  $\Delta$ , whose types are interpreted at the  $E$  level, with store  $\Gamma$  equipped with room for inserting bindings of types from  $\vec{T}$  between any two types of  $\Gamma$ .
- $\Gamma \triangleright_o^{\vec{T}} A$  denotes the translation of  $A$  at level  $o$  of the interpretation, with store  $\Gamma$  equipped with room for inserting bindings of types from  $\vec{T}$  between any two types of  $\Gamma$ .
- $\Gamma \triangleright_{\geq \vec{T}}^{\vec{X}} C$  denotes the generalization of  $\Gamma$  over  $C$ , where extensions  $\vec{X}$  of  $\vec{T}$  are used to anticipate insertions between any two types of  $\Gamma$ .

Note that the translation of types is by induction on the type, and for a type being fixed, on the interpretation level, from  $e$  to  $v$ . We abbreviate below  $\forall Y. (Y \rightarrow T) \rightarrow C$  by  $\forall Y \geq T. C$ , and similarly for  $\exists Y. (Y \rightarrow T) \wedge C$  which we abbreviate  $\exists Y \geq T. C$ .

We can now state a key lemma supporting the translation, expressing the fact that considering the translation at level  $o$  of a type  $A$  under a context  $\Gamma, A_1, \dots, A_n$  (with extensions  $\vec{T}$  inside  $\Gamma$  and  $X_0, \dots, X_n$  between the  $A_i$ ) is equivalent to considering its translation under a context  $\Gamma$  in which the right-most extension has to extend itself ( $X_0 \wedge \Gamma \triangleright_o^{\vec{T} X_0} A_1 \wedge \dots \wedge X_n$ ). Intuitively, in the last case the right-most part of the store has been glued together, and is viewed as a single extension (with a richer structure) of the left-part.



**Lemma 1** Let  $\Gamma$  a typing context and  $A_1, \dots, A_n$  a sequence of types. Let  $\vec{Y}$  be of length the length of  $\Gamma$ . Let  $A$  be a type and  $o$  a level of the hierarchy. Then, there is an isomorphism  $(\phi, \psi)$  between

$$\forall \vec{Y} X X_1 \dots X_n. (\Gamma, A_1, \dots, A_n \triangleright_o^{\vec{Y} X X_1 \dots X_n} A)$$

and

$$\forall \vec{Y} X X_1 \dots X_n. \Gamma \triangleright_o^{\vec{Y} (X \wedge (\Gamma \triangleright_t^{\vec{Y} X} A_1) \wedge X_1 \wedge \dots \wedge (\Gamma, A_1, \dots, A_{n-1} \triangleright_t^{\vec{Y} X X_1 \dots X_{n-1}} A_n) \wedge X_n)} A.$$

PROOF: The proof is by induction on  $A$ , with a subsidiary induction on the interpretation level (from  $e$  to  $v$ ). For the seek of conciseness, let us consider the case  $e$ , with  $n = 1$  and  $\Gamma = \varepsilon$ . We have to find an isomorphism between  $\forall X_0 X_1. (A_1 \triangleright_e^{X_0 X_1} A)$  and  $\forall X_0 X_1. \triangleright_e^{(X_0 \wedge (\triangleright_t^{X_0} A_1) \wedge X_1)} A$ , that is between

$$\forall X_0 X_1. (\forall Y_0 \geq X_0. Y_0 \rightarrow \triangleright_t^{Y_0} A_1 \rightarrow \forall Y_1 \geq X_1. Y_1 \rightarrow (A_1 \triangleright_t^{Y_0 Y_1} A \rightarrow \perp))$$

and

$$\forall X_0 X_1. (\forall Y \geq (X_0 \wedge (\triangleright_t^{X_0} A_1) \wedge X_1). Y \rightarrow (\triangleright_t^Y A \rightarrow \perp)).$$

From top to bottom, we instantiate  $Y_0$  by  $X_0$  and  $Y_1$  by  $X_1$ , then get and use the components of the conjunction appropriately from the knowledge of  $Y$ , and use the induction on the interpretation level  $t$  to get  $(A_1 \triangleright_t^{Y_0 Y_1} A)$  from  $(\triangleright_t^Y A)$ . From bottom to top, we instantiate  $X_0$  with  $Y_0$ ,  $X_1$  with  $Y_1$  and  $Y$  with  $Y_0 \wedge (\Gamma \triangleright_t^{Y_0} A_1) \wedge Y_1$ . It remains then to glue the components of the conjunction appropriately and use the induction on level  $t$  to conclude.  $\blacksquare$

**Lemma 2** Let  $\Gamma$  a typing context and  $A$  and  $B$  two types. We have a map weak from  $\Gamma \triangleright_E^{\vec{T}} B$  to  $\Gamma, A \triangleright_E^{\vec{T}U} B$  for all  $U$ .

Before defining the translation on terms, we introduce some operations on substitutions. Substitutions are arguments for types of the form  $\Gamma \triangleright_{\vec{X} \geq \vec{T}} C$ , hence, for  $n$  being  $|\Gamma|$  they have the form  $U_0 h_0 u_0 t_0 U_1 h_1 u_1, t_1, \dots, t_n, U_n h_n u_n$ , with each  $h_i$  a proof of  $U_i \rightarrow T_i$ , each  $u_i$  of type  $U_i$  and the  $t_i$  instantiating the types in  $\Gamma$ .

We write  $\text{id}$  for the function  $\lambda x_i. x_i$  of type  $(T_i \rightarrow T_i)$ . Let  $\tau$  be given, instantiating  $\Gamma$  intertwined with types in  $\vec{T}$ . We write  $\tau + u$  for the extension of  $\tau$  with  $u$ . It has to be intertwined with  $\vec{T}$  extended with an extra type which we take to be the unit type  $\top$ , resulting in defining  $\tau + u \triangleq \tau u \top \text{id}()$ .

We consider also a caching operation. For that purpose, let us decompose some  $\tau$  as above into  $X_0 h_0 u_0 t_0 X_1 h_1 u_1 \dots X_{n-1} h_{n-1} u_{n-1} t_n X_n h_n u_n$  where the  $t_i$  are typed derived from the types in  $\Gamma$  and each  $u_i$  is typed in  $X_i$  which extends  $T_i$  using a proof  $h_i : X_i \rightarrow T_i$ . Let us write  $\tau$  for  $X_0 h_0 u_0 t_0 X_1 h_1 u_1 \dots t_i$  and  $\tau'$  for  $X_i h_i u_i \dots X_{n-1} h_{n-1} u_{n-1} t_n X_n h_n u_n$ . Then, we write  $\tau @ (\tau')$  for the substitution obtained by gluing all of  $X_i h_i u_i \dots X_{n-1} h_{n-1} u_{n-1} t_n X_n h_n u_n$  into a single component extending  $T_i$ , as in Proposition 1.

We can then define the translation of terms as given in Figure 6. Note that lemmas 1 and 2 are used in the cases  $\llbracket x_i \rrbracket_v^{\vec{T}}$  and  $\llbracket \lambda x_n. t \rrbracket_v^{\vec{T}}$ . We also assume given in System F a free variable  $\alpha$  of type  $\Gamma \triangleright_v^{\vec{T}} A$  for each instance of a constant  $\alpha$  typed by  $\Gamma; \alpha : A \vdash \Delta$  in  $\bar{\lambda}_{[v\tau\star]}$ .

**Theorem 3** The translation is well-typed, i.e.

$$\begin{array}{ll} \Gamma \vdash_v \Delta; v : A & \text{implies} \quad \llbracket \Gamma \vdash_v \Delta; v : A \rrbracket^{\vec{T}} \\ \Gamma; F : A \vdash_F \Delta & \text{implies} \quad \llbracket \Gamma; F : A \vdash_F \Delta \rrbracket^{\vec{T}} \\ \Gamma \vdash_v \Delta; V : A & \text{implies} \quad \llbracket \Gamma \vdash_v \Delta; V : A \rrbracket^{\vec{T}} \\ \Gamma; E : A \vdash_E \Delta & \text{implies} \quad \llbracket \Gamma; E : A \vdash_E \Delta \rrbracket^{\vec{T}} \\ \Gamma \vdash_t \Delta; t : A & \text{implies} \quad \llbracket \Gamma \vdash_t \Delta; t : A \rrbracket^{\vec{T}} \\ \Gamma; e : A \vdash_e \Delta & \text{implies} \quad \llbracket \Gamma; e : A \vdash_e \Delta \rrbracket^{\vec{T}} \\ c : (\Gamma \vdash_c \Delta) & \text{implies} \quad \llbracket c : (\Gamma \vdash_c \Delta) \rrbracket^{\vec{T}} \end{array}$$

$[[\lambda x_n.t]]_v^{\vec{T}} \tau u E$	$\triangleq$	$[[t]]_t^{\vec{T}\top} (\tau + u) (\text{weak } E)$
$[[\alpha]]_F^{\vec{T}} \tau v$	$\triangleq$	$\alpha \tau v$
$[[t \cdot E]]_F^{\vec{T}} \tau v$	$\triangleq$	$v \tau [[t]]_t^{\vec{T}} [[E]]_E^{\vec{T}}$
$[[x_i]]_v^{\vec{T}} \tau t_i \tau' F$	$\triangleq$	$t_i \tau (\lambda \tau. \lambda V. V \tau @ \langle (\lambda \tau. \lambda E. E \tau V) \tau' \rangle (\psi F))$
$[[v]]_V^{\vec{T}} \tau F$	$\triangleq$	$F \tau [[v]]_v^{\vec{T}}$
$[[F]]_E^{\vec{T}} \tau V$	$\triangleq$	$V \tau [[F]]_F^{\vec{T}}$
$[[\alpha_i]]_E^{\vec{T}} \tau V$	$\triangleq$	$\alpha_i \tau V$
$[[\tilde{\mu}[x_i]. \langle x_i    F \rangle \tau']]_E^{\vec{T}} \tau V$	$\triangleq$	$V \tau @ \langle (\lambda \tau. \lambda E. E \tau V) \tau' \rangle (\psi [[F]]_F^{\vec{T}})$
$[[V]]_t^{\vec{T}} \tau E$	$\triangleq$	$E \tau [[V]]_V^{\vec{T}}$
$[[\mu \alpha_n.c]]_t^{\vec{T}} \tau E$	$\triangleq$	$\lambda \alpha_n. ([c]]_c^{\vec{T}} \tau) E$
$[[E]]_e^{\vec{T}} \tau t$	$\triangleq$	$t \tau [[E]]_E^{\vec{T}}$
$[[\tilde{\mu} x_n.c]]_e^{\vec{T}} \tau t$	$\triangleq$	$[[c]]_c^{\vec{T}} \tau t$
$[[\langle t    e \rangle]]_c^{\vec{T}} \tau$	$\triangleq$	$[[e]]_e^{\vec{T}} \tau [[t]]_t^{\vec{T}}$

Figure 6: Translation of terms

**Corollary 4** *The above continuation-passing-style defines a semantics of call-by-need  $\lambda$ -calculus with control which is strongly normalizing in the simply-typed case.*

This is to be contrasted with Okasaki-Lee-Tarditi’s semantics which is not normalizing, as shown in Ariola *et al* [2].

## Acknowledgments

The first author is thanking Keiko Nakata, José Carlos Espírito Santo, Luís Pinto, Zena Ariola, Paul Downen, Alexis Saurin and Rossen Mikhov for initial discussions on typing the continuation-passing-style semantics of call-by-need  $\lambda$ -calculus.

## References

- [1] Zena Ariola & Matthias Felleisen (1993): *The Call-By-Need Lambda Calculus*. *J. Funct. Program.* 7(3), pp. 265–301.
- [2] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata & Alexis Saurin (2012): *Classical call-by-need sequent calculi: The unity of semantic artifacts*. In: *Fuji International Symposium on Functional and Logic Programming (FLOPS '12), Kobe, Japan, May 23-25, 2012*. To appear.
- [3] Zena M. Ariola, Hugo Herbelin & Amr Sabry (2004): *A Type-Theoretic Foundation of Continuations and Prompts*. In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snowbird, UT, USA, September 19-21, 2004*, ACM Press, New York, pp. 40–53.
- [4] Zena M. Ariola, Hugo Herbelin & Alexis Saurin (2011): *Classical Call-by-Need and Duality*. In C.-H. Luke Ong, editor: *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings, Lecture Notes in Computer Science 6690*, Springer, pp. 27–44. Available at [http://dx.doi.org/10.1007/978-3-642-21691-6\\_6](http://dx.doi.org/10.1007/978-3-642-21691-6_6).
- [5] Tristan Crolard (1999): *A confluent lambda-calculus with a catch/throw mechanism*. *J. Funct. Program.* 9(6), pp. 625–647.
- [6] Pierre-Louis Curien & Hugo Herbelin (2000): *The duality of computation*. In: *Proceedings of ICFP 2000, SIGPLAN Notices 35(9)*, ACM, pp. 233–243.
- [7] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker & Bruce F. Duba (1986): *Reasoning with continuations*. In: *First Symposium on Logic and Computer Science*, pp. 131–141.
- [8] Hugo Herbelin (2005): *C’est maintenant qu’on calcule: au cœur de la dualité*. Habilitation thesis, University Paris 11.
- [9] Jean-Louis Krivine (2004): *Realizability in classical logic. Panoramas et synthèses*. To appear.
- [10] John Maraist, Martin Odersky & Philip Wadler (1998): *The Call-by-Need Lambda Calculus*. *J. Funct. Program.* 8(3), pp. 275–317.
- [11] Eugenio Moggi (1988): *Computational lambda-calculus and monads*. Technical Report ECS-LFCS-88-66, Edinburgh Univ.
- [12] Chris Okasaki, Peter Lee & David Tarditi (1994): *Call-by-Need and Continuation-Passing Style*. *Lisp and Symbolic Computation* 7(1), pp. 57–82.
- [13] Michel Parigot (1991): *Free Deduction: An Analysis of "Computations" in Classical Logic*. In Andrei Voronkov, editor: *Proceedings of LPAR, LNCS 592*, Springer, pp. 361–380.
- [14] Gordon D. Plotkin (1975): *Call-by-name, call-by-value and the lambda-calculus*. *Theor. Comput. Sci.* 1, pp. 125–159.

- [15] Amr Sabry & Matthias Felleisen (1993): *Reasoning about Programs in Continuation-Passing Style*. *Lisp and Symbolic Computation* 6(3-4), pp. 289–360.