A continuation-passing-style interpretation of simply-typed call-by-need λ-calculus with control within System F

Hugo Herbelin, Étienne Miquey

A continuation-passing-style interpretation of simply-typed call-by-need λ-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 λ-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 α-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 the λ-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.

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.

Call-by-need evaluation strategy is a third evaluation strategy of the λ-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. It has in common with the call-by-value evaluation strategy 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 particular, 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 the λ-calculus based on the operational rule β and the extensional rule η.

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

Submitted to:

by Moggi [11] with the convenient introduction of a native `let` operator[1]. 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 the λ-calculus.

For call-by-need, a continuation-passing-style semantics was proposed by Okasaki-Lee-Tarditi [12] but this semantics does not ensure normalization of simply-typed call-by-need evaluation, as shown in [2], thus failing to ensure a property which however holds in the simply-typed call-by-name and call-by-value cases.

Continuation-passing-style semantics de facto gives a semantics to the extension of the calculus with control operators, i.e. with operators such as Scheme's `callcc`, Felleisen's $\mathscr{C}$, $\mathscr{K}$, or $\mathscr{A}$ operators [7], Parigot's μ 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 λ-calculus, their different intentional behavior induces different continuation-passing-style semantics, and this reflects that they behave observationally differently when control operators are considered.

Building on top of the duality between programs and their evaluation contexts [6], and the duality between the `let` construct (which binds programs) and a control operator such as Parigot's μ (which binds evaluation contexts), the first author proposed the core of a call-by-need reduction semantics supporting control operators [8]. Let us consider the following language with term constants ranged over by c and context constants ranged over by ξ:

$$\begin{array}{llcl@{\qquad}llcl}
\text{Strong values} & W & ::= & \lambda x.t \mid \boldsymbol{x} & \text{Strong contexts} & F & ::= & t\cdot e \mid \boldsymbol{\alpha} \\
\text{Weak values} & V & ::= & W \mid x & \text{Weak contexts} & E & ::= & F \mid \alpha \\
\text{Terms} & v & ::= & V \mid \mu\alpha.c & \text{Evaluation contexts} & e & ::= & E \mid \tilde{\mu}x.c
\end{array}$$

$$\text{Commands } c \ ::= \ \langle v \| e \rangle$$

with the following reduction rules parameterized over a sets of terms $\mathscr{V}$ and a set of evaluation contexts $\mathscr{E}$:

$$\begin{array}{lcll}
\langle v \| \tilde{\mu}x.c \rangle & \to & c[v/x] & v \in \mathscr{V} \\
\langle \mu\alpha.c \| e \rangle & \to & c[e/\alpha] & e \in \mathscr{E} \\
\langle \lambda x.v \| v' \cdot e \rangle & \to & \langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle &
\end{array}$$

Then, the difference between call-by-name, call-by-value and call-by-need can be characterized by how the critical pair

$$\begin{array}{ccc}
 & \langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle & \\
\swarrow & & \searrow \\
c[\tilde{\mu}x.c'/\alpha] & & c'[\mu\alpha.c/x]
\end{array}$$

[1] 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 λ 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 $\to$ to internalize derivability $\vdash$ leading to require $\to$ 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.

is solved, which amounts to provide with two $\mathscr{V}$ and $\mathscr{E}$ such that the two rules do not overlap:

- Call-by-name: $\mathscr{V}$ = Terms, $\mathscr{E}$ = Weak contexts
- Call-by-value: $\mathscr{V}$ = Weak values, $\mathscr{E}$ = Evaluation contexts
- Call-by-need: $\mathscr{V}$ = Strong values, $\mathscr{E}$ = Weak contexts $\cup$ Forcing contexts, where forcing contexts are expressions of the form $\tilde{\mu}x.C[\langle x \| F\rangle]$ where C, called suspension, is a command with a hole as defined by the grammar
$$C[\,] \quad ::= \quad [\,] \mid \langle \mu\alpha.c \| \tilde{\mu}x.C[\,]\rangle$$
In particular, forcing contexts are those evaluation contexts whose evaluation is blocked on the knowledge of x, hence requiring the evaluation of what is bound to x. Also, suspensions are those commands which stack instances of the $\langle v \| e\rangle$ expression for which neither v is in $\mathscr{V}$ (meaning it is some $\mu\alpha.c$) nor e in $\mathscr{E}$ (meaning it is a $\tilde{\mu}x.c$ which is not a forcing context).

This semantics was studied in Ariola *et al* [2] eventually providing a continuation-passing-style semantics[2]. It is this semantics that we study in this paper.

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 $\overline{\lambda}_{[lv\tau\star]}$, even though the treatment could be done for the simpler calculus $\overline{\lambda}_{lv}$ (Section 1) as well. The calculi $\overline{\lambda}_{[lv\tau\star]}$ is a sequent calculus targeted on call-by-need. We recall its syntax in Section 2 before equipping it with a system of simple types in Section 3. The core of the paper is in typing the continuation-passing-style translation, what is done in Section 4.

1 The $\bar{\lambda}_{lv}$-calculus

We first recall the syntax and typing rules of the $\bar{\lambda}_{lv}$-calculus [2], that is a call-by-need adaptation of the $\bar{\lambda}\mu\tilde{\mu}$-calculus [6].

$$\begin{array}{llcl@{\qquad}llcl}
\text{Commands} & c & ::= & \langle t \| e\rangle & \text{Evaluation contexts} & e & ::= & E \mid \tilde{\mu}x.c \\
\text{Terms} & t & ::= & V \mid \mu\alpha.c & \text{Catchable contexts} & E & ::= & F \mid \alpha \mid \tilde{\mu}x.C[\langle x \| F\rangle] \\
\text{Values} & V & ::= & \lambda x.t \mid x & \text{Forcing contexts} & F & ::= & \boldsymbol{\alpha} \mid t \cdot E
\end{array}$$

$$\text{Meta-contexts} \quad C \quad ::= \quad \square \mid \langle \mu\alpha.c \| \tilde{\mu}x.C\rangle$$

The λ_{lv} reduction, written as $\to_{lv}$, denotes the compatible closure of the rules:

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

[2] A similar semantics was previously studied in [4] with $\mathscr{E}$ defined instead to be $\tilde{\mu}x.C[\langle x|E\|]\rangle$ (with same definition of $\mathscr{C}$) and a definition of $\mathscr{V}$ which was different whether $\tilde{\mu}x.c$ was a forcing context ($\mathscr{V}$ was then the strong values) or not ($\mathscr{V}$ was then the weak values). Another variant is discussed in Section 6 of [2] where $\mathscr{E}$ similarly defined to be $\tilde{\mu}x.C[\langle x|E\|]\rangle$ and $\mathscr{V}$ to be (uniformly) the strong values. All three semantics seem to make sense to us. Note that term constant are not considered in [2] nor [4]. We add them here for symmetry of the presentation.

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

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

$$\frac{\Gamma\vdash t:A\mid\Delta\qquad\Gamma\mid e:A\vdash\Delta}{\langle t\|e\rangle:(\Gamma\vdash\Delta)}\qquad\frac{}{\Gamma\mid\boldsymbol{\alpha}:A\vdash\Delta}$$

Figure 1: Typing rules for $\bar{\lambda}_{lv}$

A *forcing* contexts, which is either a stack $t\cdot E$ or a co-constant $\boldsymbol{\alpha}$, eagerly demands a value, and drives the computation forward. A variable is said to be *needed* or *demanded* if it is in a command with a forcing context, as in $\langle x\|F\rangle$. Furthermore, in a $\tilde{\mu}$-binding of the form $\tilde{\mu}x.C[\langle x\|F\rangle]$, we say that the bound variable x has been *forced*. The $C[\;]$ is a meta-context, which identifies the standard redex in a command. 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 typing rules (see Figure 1) for the $\bar{\lambda}_{lv}$-calculus are the usual rules of the classical sequent calculus [6].

2 The $\bar{\lambda}_{[lv\tau]}$-calculus syntax

While all the results that are presented in the sequel of this paper could be directly expressed using the $\bar{\lambda}_{lv}$-calculus. the continuation-passing-style translation we present naturally arises from the decomposition of this calculus into a small-step one, the $\bar{\lambda}_{[lv\tau]}$-calculus. 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.

We now recall the syntax of $\overline{\lambda}_{[lv\tau\star]}$-calculus from [2]. This calculus enjoys small-step reduction rules, which makes it closer from an abstract machine. In particular, it uses an explicit *environment* τ binding terms to variables (we alternatively call it a *substitution*), where terms are lazily stored by default, and which allows to have a head-reduction system.

We introduce 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 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 α_i, where, again, α is a fixed name indicating that the variable is binding evaluation contexts.Note that binders, substituting a term t within another term requires in general to renumber the bound variables of t, shifting them by the number of binders traversed by t before reaching the positions where it is substituted.

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

Figure 2: Reduction rules of the $\bar{\lambda}_{[lv\tau]}$-calculus

Finally, we introduce a new type of catchable contexts, $\tilde{\mu}[x_i].\langle x_i \| F \rangle$, expressing the fact that the variable x_i is forced at top-level. The syntax of the language is given by:

$$
\begin{array}{llll}
\text{Closures} & l & ::= & c\tau \\
\text{Commands} & c & ::= & \langle t \| e \rangle
\end{array}
\qquad
\begin{array}{llll}
\text{Substitutions} & \tau & ::= & \varepsilon \mid \tau[x_i := t]
\end{array}
$$

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

and the reduction rules are given in Figure 2.

The different syntactic categories can be understood as the different levels of alternation in a context-free abstract machine [2]: the priority is first given to context of level e (lazy storage of terms), then to terms at level p (evaluation of $\mu\alpha$ into values), then back to contexts at level E and so on until level F. These different categories are directly reflected in the definition of 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 in the next Section.

3 Typing rules

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: one of the letters $v, V, t, F, E, e, l, c, \tau$. 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 and closures 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 nine kinds of sequents:

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

Types and typing contexts are defined by:

$$
A, B \quad ::= \quad X \mid A \rightarrow B
\qquad\qquad
\begin{array}{lll}
\Gamma & ::= & \varepsilon \mid \Gamma, x : A \\
\Delta & ::= & \varepsilon \mid \Delta, \alpha : A
\end{array}
$$

The typing rules are given on Figure 3 where $|\Gamma|$ denotes the length of Γ, and $\Gamma(i)$ for $i < |\Gamma|$ denotes the i^{th} in Γ, and similarly for $|\Delta|$ and $\Delta(i)$.

$$\frac{\Gamma, x_n : A \vdash_t t : B \mid \Delta \qquad |\Gamma| = n}{\Gamma \vdash_v \lambda x_n.t : A \to B \mid \Delta}$$

$$\frac{\Gamma(i) = x_i : A}{\Gamma \vdash_V x_i : A \mid \Delta} \qquad \frac{\Gamma \vdash_v v : A \mid \Delta}{\Gamma \vdash_V v : A \mid \Delta}$$

$$\frac{\Gamma \vdash_V V : A \mid \Delta}{\Gamma \vdash_t V : A \mid \Delta} \qquad \frac{c : (\Gamma \vdash_c \Delta, \alpha_n : A) \qquad |\Delta| = n}{\Gamma \vdash_t \mu\alpha_n.c : A \mid \Delta}$$

$$\frac{}{\Gamma \mid \boldsymbol{\alpha} : A \vdash_F \Delta} \qquad \frac{\Gamma \vdash_t t : A \mid \Delta \qquad \Gamma \mid E : B \vdash_E \Delta}{\Gamma \mid t \cdot E \vdash_F \Delta}$$

$$\frac{\Delta(i) = \alpha_i : A}{\Gamma \mid \alpha_i : A \vdash_E \Delta} \quad \frac{\Gamma \mid F : A \vdash_F \Delta}{\Gamma \mid F : A \vdash_E \Delta} \quad \frac{\Gamma, x_n : A, \Gamma'; F : A \vdash_F \Delta \quad \Gamma \vdash \tau : \Gamma' \mid \Delta \quad |\Gamma| = n}{\Gamma \mid \tilde{\mu}[x_n].\langle x_n \| F\rangle\tau : A \vdash_E \Delta}$$

$$\frac{\Gamma \mid E : A \vdash_E \Delta}{\Gamma \mid E : A \vdash_e \Delta} \qquad \frac{c : (\Gamma, x_n : A \vdash_t \Delta) \quad |\Gamma| = n}{\Gamma \mid \tilde{\mu} x_n.c : A \vdash_e \Delta}$$

$$\frac{\Gamma \vdash_t t : A \mid \Delta \qquad \Gamma \mid e : A \vdash_e \Delta}{\langle t \| e\rangle : (\Gamma \vdash_c \Delta)} \qquad \frac{c : (\Gamma, \Gamma' \vdash_c \Delta) \qquad \Gamma \vdash_\tau \tau : \Gamma' \mid \Delta}{c\tau : (\Gamma \vdash_l \Delta)}$$

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

Figure 3: Typing rules

4 A typed cps-translation

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

We shall first give a translation of typing sequents for $\overline{\lambda}_{[lv\tau\star]}$ into typing sequents of System F, whose syntax and typing rules “à la Church” are recalled on Figure 4. In System F, one can define a unit type $\top \triangleq \forall X.X \to X$ with canonical inhabitant $() \triangleq \Lambda X.\lambda x.x$. Similarly, one can define a n-ary conjunction $T_1 \wedge \ldots \wedge T_n \triangleq \forall X.(T_1 \to \ldots \to T_n \to X) \to X$ with constructor $\langle t_1, \ldots, t_n\rangle \triangleq \Lambda X.\lambda x.x t_1 \ldots t_n$. Projections can then be defined as well. Finally, we define $\perp \triangleq \forall X.X$.

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

Syntax

$$
\begin{array}{llll}
\text{Types} & T,U & ::= & X \mid T \to U \mid \forall X.T \\
\text{Terms} & t,u & ::= & x \mid \lambda x.t \mid t\,u \mid \Lambda X.t \mid t\,T \\
\text{Typing contexts} & \Gamma & ::= & \varepsilon \mid \Gamma, x : T \mid \Gamma, X
\end{array}
$$

where X ranges over countably many type variables

Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x.t : U \to T} \qquad \frac{\Gamma \vdash t : U \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\,u : T}$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T} \qquad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t\,U : T[X := U]}$$

Figure 4: Syntax and typing rules of System F presented à la Church

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 $\overline{\lambda}_{[lv\tau\star]}$ sequent. Then, the translation of a sequent such as $\Gamma \vdash_t A \mid \Delta$ is a System F sequent $\overset{3}{\neg}\,\Delta, \overset{4}{\neg}\,\Gamma \vdash \overset{4}{\neg}\,A$, using the notation $\overset{n+1}{\neg} A \triangleq \overset{n}{\neg}(A)$. There are respectively 3 negations over Δ and 4 over Γ and A because, as shown in [2] and as emphasized by the 6 nested syntactic categories used to define $\overline{\lambda}_{[lv\tau\star]}$, there are 6 levels of control in call-by-need, leading to 6 mutually defined levels of interpretation:

- $[\![A]\!]_v$ for strong values: a strong value of type $A \to B$ is interpreted as a term of type $[\![A \to B]\!]_v \triangleq [\![A]\!]_t \to [\![B]\!]_E \to \bot$, i.e., informally, $\overset{4}{\neg}\,A \to \overset{3}{\neg}\,B \to \bot$.
- $[\![A]\!]_F$ 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 $[\![A]\!]_F \triangleq [\![A]\!]_v \to \bot$, i.e., informally, $\neg A$.
- $[\![A]\!]_V$ 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 $[\![A]\!]_V \triangleq [\![A]\!]_F \to \bot$, i.e., informally $\overset{2}{\neg}\,A$.
- $[\![A]\!]_E$ for catchable evaluation contexts: a catchable evaluation context is expecting to take control over a weak value, possibly duplicating or erasing it with the $\tilde{\mu}[x].\langle x \| F \rangle \tau$ construct, so $[\![A]\!]_E \triangleq [\![A]\!]_V \to \bot$, i.e., informally $\overset{3}{\neg}\,A$.
- $[\![A]\!]_t$ 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 $[\![A]\!]_t \triangleq [\![A]\!]_E \to \bot$, i.e., informally $\overset{4}{\neg}\,A$.
- $[\![A]\!]_e$ 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 $[\![A]\!]_e \triangleq [\![A]\!]_t \to \bot$, i.e., informally $\overset{5}{\neg}\,A$.

In particular, when translating a sequent such as $\Gamma \vdash_t A \mid \Delta$, the context Δ is interpreted as a context of catchable evaluation contexts while Γ 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 $\Gamma \vdash_t A \mid \Delta$ the translation is $[\![\Delta]\!]_E, [\![\Gamma]\!]_t \vdash [\![A]\!]_t$, and similarly for other levels, e.g., $\Gamma \mid A \vdash_e \Delta$ translates to $[\![\Delta]\!]_E, [\![\Gamma]\!]_t \vdash [\![A]\!]_e$.

The continuation-passing-style part being settled, the environment-passing-style part should be considered. In particular, the translation of $\Gamma \vdash_t A \mid \Delta$ is not anymore a sequent $[\![\Delta]\!]_E, [\![\Gamma]\!]_t \vdash [\![A]\!]_t$ but instead a sequent roughly of the form $[\![\Delta]\!]_E \vdash [\![\Gamma]\!]_t \to [\![A]\!]_t$, with actually Γ being passed around not only at the top level of $[\![A]\!]_t$ but also every time a negation is used. Additionally, abstraction over Γ should be passed around over all types in Δ. And, moreover, the translation of each type in Γ 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 $[\![X_1]\!]_t, [\![X_2]\!]_t, [\![Y]\!]_E \vdash [\![X]\!]_V$, where:

- $[\![X_1]\!]_t$ is $\overset{4}{\neg} X_1$,
- $[\![X_2]\!]_t$, dependent on $[\![X_1]\!]_t$ at each level of negation is $[\![X_1]\!]_t \to \neg([\![X_1]\!]_t \to \neg([\![X_1]\!]_t \to \neg([\![X_1]\!]_t \to \neg X_2)))$,
- $[\![Y]\!]_E$, similarly dependent on $[\![X_1]\!]_t$ and $[\![X_2]\!]_t$ at each level is $([\![X_1]\!]_t \to [\![X_2]\!]_t \to \neg([\![X_1]\!]_t \to [\![X_2]\!]_t \to \neg([\![X_1]\!]_t \to [\![X_2]\!]_t \to \neg Y)))$,
- $[\![X]\!]_V$, also dependent on $[\![X_1]\!]_t$ and $[\![X_2]\!]_t$ at each level is $[\![X_1]\!]_t \to [\![X_2]\!]_t \to \neg([\![X_1]\!]_t \to [\![X_2]\!]_t \to \neg X)$.

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 $T_0, A_1, T_1, ..., A_n, T_n$ for arbitrary T_0, ..., T_n, whenever the typing context is $A_1, ..., A_n$, 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 5 and Figure 6 where:

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

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 $\forall Y.(Y \to T) \to C$ by $\forall Y \geq T.C$.

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 $\Gamma, A_1, \ldots, A_n$ (with extensions $\vec{T}$ inside Γ and $X_0, \ldots, X_n$ between the A_i) is equivalent to considering its translation under a context Γ in which the right-most extension has to extend itself $(X_0 \wedge \Gamma \triangleright_o^{\vec{T}X_0} A_1 \wedge \ldots \wedge X_n)$. 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 Γ be a typing context and A_1, ..., A_n a sequence of types. Let $\vec{Y}$ be of length the length of Γ. Let A be a type and o a level of the hierarchy. Then, there is an isomorphism (ϕ, ψ) between*

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

$$
\begin{array}{lcl}
[\![\Gamma \mid e : A \vdash_e \Delta]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![e]\!]_e^{\vec{T}} : \Gamma \rhd_e^{\vec{T}} A \\
[\![\Gamma \vdash_t t : A \mid \Delta]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![t]\!]_t^{\vec{T}} : \Gamma \rhd_g^{\vec{T}} A \\
[\![\Gamma \mid E : A \vdash_E \Delta]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![E]\!]_E^{\vec{T}} : \Gamma \rhd_E^{\vec{T}} A \\
[\![\Gamma \vdash_V V : A \mid \Delta]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![V]\!]_V^{\vec{T}} : \Gamma \rhd_V^{\vec{T}} A \\
[\![\Gamma \mid F : A \vdash_F \Delta]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![F]\!]_F^{\vec{T}} : \Gamma \rhd_F^{\vec{T}} A \\
[\![\Gamma \vdash_v v : A \mid \Delta]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![v]\!]_v^{\vec{T}} : \Gamma \rhd_v^{\vec{T}} A \\
[\![c : (\Gamma \vdash_c \Delta)]\!]^{\vec{T}} & \triangleq & \Gamma \rhd^{\vec{T}} \Delta \vdash [\![c]\!]_c^{\vec{T}} : (\Gamma \rhd^{\vec{X} \geq \vec{T}} \bot)
\end{array}
$$

Figure 5: Translation of judgments

$$
\begin{array}{lcl}
\Gamma \rhd_e^{\vec{T}} A & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_t^{\vec{X}} A \to \bot) \\
\Gamma \rhd_t^{\vec{T}} A & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_E^{\vec{X}} A \to \bot) \\
\Gamma \rhd_E^{\vec{T}} A & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_V^{\vec{X}} A \to \bot) \\
\Gamma \rhd_V^{\vec{T}} A & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_F^{\vec{X}} A \to \bot) \\
\Gamma \rhd_F^{\vec{T}} A & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_v^{\vec{X}} A \to \bot) \\
\Gamma \rhd_v^{\vec{T}} A \to B & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_t^{\vec{X}} A \to \Gamma \rhd_E^{\vec{X}} B \to \bot) \\
\Gamma \rhd_v^{\vec{T}} Y & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} Y \\
 & & \\
\varepsilon \rhd^{Y \geq U} B(Y) & \triangleq & \forall Y \geq U. Y \to B(Y) \\
\Gamma, x : A \rhd^{\vec{X} \geq \vec{T}, Y \geq U} B(\vec{X}, Y) & \triangleq & \Gamma \rhd^{\vec{X} \geq \vec{T}} (\Gamma \rhd_t^{\vec{X}} A \to \forall Y \geq U. Y \to B(\vec{X}, Y)) \\
 & & \\
\Gamma \rhd^{\vec{T}} \varepsilon & \triangleq & \varepsilon \\
\Gamma \rhd^{\vec{T}} (\Delta, \alpha : A) & \triangleq & (\Gamma \rhd^{\vec{T}} \Delta), (\Gamma \rhd_E^{\vec{T}} A)
\end{array}
$$

Figure 6: Translation of types

and

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

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 $\forall X_0 X_1. (A_1 \rhd_e^{X_0 X_1} A)$ and $\forall X_0 X_1. \rhd_e^{(X_0 \wedge (\rhd_t^{X_0} A_1) \wedge X_1} A$, that is between

$$\forall X_0 X_1. (\forall Y_0 \geq X_0. Y_0 \to \rhd_t^{Y_0} A_1 \to \forall Y_1 \geq X_1. Y_1 \to (A_1 \rhd_t^{Y_0 Y_1} A \to \bot))$$

and

$$\forall X_0 X_1. (\forall Y \geq (X_0 \wedge (\rhd_t^{X_0} A_1) \wedge X_1). Y \to (\rhd_t^{Y} A \to \bot)).$$

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

$$
\begin{array}{lcl}
[\![\lambda x_n.t]\!]_v^{\vec{T}}\,\tau\,u\,E & \triangleq & [\![t]\!]_t^{\vec{T}\top}\,(\tau+u)\,(\mathsf{weak}\,E) \\[1em]
[\![\boldsymbol{\alpha}]\!]_F^{\vec{T}}\,\tau\,v & \triangleq & \boldsymbol{\alpha}\,\tau\,v \\
[\![t\cdot E]\!]_F^{\vec{T}}\,\tau\,v & \triangleq & v\,\tau\,[\![t]\!]_t^{\vec{T}}\,[\![E]\!]_E^{\vec{T}} \\[1em]
[\![x_i]\!]_V^{\vec{T}}\,\tau\,t_i\,\tau'\,F & \triangleq & t_i\,\tau\,(\lambda\tau.\lambda V.V\,\tau@\langle(\lambda\tau.\lambda E.E\,\tau\,V)\,\tau'\rangle\,(\psi F)) \\
[\![v]\!]_V^{\vec{T}}\,\tau\,F & \triangleq & F\,\tau\,[\![v]\!]_v^{\vec{T}} \\[1em]
[\![F]\!]_E^{\vec{T}}\,\tau\,V & \triangleq & V\,\tau\,[\![F]\!]_F^{\vec{T}} \\
[\![\alpha_i]\!]_E^{\vec{T}}\,\tau\,V & \triangleq & \alpha_i\,\tau\,V \\
[\![\tilde{\mu}[x_i].\langle x_i \| F\rangle\tau']\!]_E^{\vec{T}}\,\tau\,V & \triangleq & V\,\tau@\langle(\lambda\tau.\lambda E.E\,\tau\,V)\,\tau')\rangle\,(\psi[\![F]\!]_F^{\vec{T}}) \\[1em]
[\![V]\!]_t^{\vec{T}}\,\tau\,E & \triangleq & E\,\tau\,[\![V]\!]_V^{\vec{T}} \\
[\![\mu\alpha_n.c]\!]_t^{\vec{T}}\,\tau\,E & \triangleq & \lambda\alpha_n.([\![c]\!]_c^{\vec{T}}\,\tau)\,E \\[1em]
[\![E]\!]_e^{\vec{T}}\,\tau\,t & \triangleq & t\,\tau\,[\![E]\!]_E^{\vec{T}} \\
[\![\tilde{\mu}x_n.c]\!]_e^{\vec{T}}\,\tau\,t & \triangleq & [\![c]\!]_c^{\vec{T}}\,\tau\,t \\[1em]
[\![\langle t\|e\rangle]\!]_c^{\vec{T}}\,\tau & \triangleq & [\![e]\!]_e^{\vec{T}}\,\tau\,[\![t]\!]_t^{\vec{T}}
\end{array}
$$

Figure 7: Translation of terms

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

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

We write id for the function $\lambda x_i.x_i$ of type $(T_i \to T_i)$. Let τ be given, instantiating Γ intertwined with types in $\vec{T}$. We write $\tau+u$ for the extension of τ 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 $\tau+u \triangleq \tau\,u\,\top\,\mathsf{id}\,()$.

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

We can then define the translation of terms as given in Figure 7. Note that lemmas 1 and 2 are used in the cases $[\![x_i]\!]_V^{\vec{T}}$ and $[\![\lambda x_n.t]\!]_v^{\vec{T}}$. We also assume given in System F a free variable $\boldsymbol{\alpha}$ of type $\Gamma \rhd_v^{\vec{T}} A$ for each instance of a constant $\boldsymbol{\alpha}$ typed by $\Gamma \mid \boldsymbol{\alpha} : A \vdash \Delta$ in $\overline{\lambda}_{[lv\tau\star]}$.

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

$$\begin{array}{lll}
\Gamma \vdash_v v : A \mid \Delta & \textit{implies} & [\![\Gamma \vdash_v v : A \mid \Delta]\!]^{\vec{T}} \\
\Gamma \mid F : A \vdash_F \Delta & \textit{implies} & [\![\Gamma \mid F : A \vdash_F \Delta]\!]^{\vec{T}} \\
\Gamma \vdash_V V : A \mid \Delta & \textit{implies} & [\![\Gamma \vdash_V V : A \mid \Delta]\!]^{\vec{T}} \\
\Gamma \mid E : A \vdash_E \Delta & \textit{implies} & [\![\Gamma \mid E : A \vdash_E \Delta]\!]^{\vec{T}} \\
\Gamma \vdash_t t : A \mid \Delta & \textit{implies} & [\![\Gamma \vdash_t t : A \mid \Delta]\!]^{\vec{T}} \\
\Gamma \mid e : A \vdash_e \Delta & \textit{implies} & [\![\Gamma \mid e : A \vdash_e \Delta]\!]^{\vec{T}} \\
c : (\Gamma \vdash_c \Delta) & \textit{implies} & [\![c : (\Gamma \vdash_c \Delta)]\!]^{\vec{T}}
\end{array}$$

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

This is to be contrasted with Okasaki, Lee and 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 λ-calculus.

References

[1] Zena Ariola & Matthias Felleisen (1993): *The Call-By-Need Lambda Calculus*. *J. Funct. Program.* 7(3), pp. 265–301, doi:dx.doi.org/10.1017/S0956796897002724.

[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 Tom Schrijvers & Peter Thiemann, editors: *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, *Lecture Notes in Computer Science* 7294, Springer, pp. 32–46, doi:10.1007/978-3-642-29822-6. Available at `http://dx.doi.org/10.1007/978-3-642-29822-6`.

[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, doi: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, doi:10.1017/S0956796899003512.

[6] Pierre-Louis Curien & Hugo Herbelin (2000): *The duality of computation*. In: *Proceedings of ICFP 2000*, SIGPLAN Notices 35(9), ACM, pp. 233–243, doi:10.1145/351240.351262.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker & Bruce F. Duba (1986): *Reasoning with Continuations*. In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, 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., doi:10.1109/LICS.1989.39155.

[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. Available at `http://dx.doi.org/10.1007/3-540-55460-2_27`.

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

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