Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Revisiting the CPS Transformation and its Implementation

François Pottier

Received: date / Accepted: date

Abstract We give a machine-checked definition and proof of semantic correctness for Danvy and Filinski's properly tail-recursive, one-pass, call-by-value CPS transformation. We do so in the setting of the pure λ-calculus extended with a let construct. We propose a new first-order, one-pass, compositional formulation of the transformation. We point out that Danvy and Filinski's simulation diagram does not hold in the presence of let, and prove a slightly more complex diagram, which involves parallel reduction. We represent variables as de Bruijn indices and show that, given the current state of the art, this does not represent a significant impediment to formalization. Finally, we note that, given this representation of terms, it is not obvious how to efficiently implement the transformation. To address this issue, we propose a novel higher-order formulation of the transformation. We prove that it is correct and informally argue that it runs in time $O(n \log n)$.

1 Introduction

The transformation of call-by-value λ-terms into continuation-passing style, independently discovered by many researchers (Reynolds, 1993) and first explicitly formulated by Fischer (1972, 1993), was first proved correct by Plotkin (1975).

Plotkin's CPS transformation produces many "administrative" redexes, which are a source of inefficiency and complicate the proof of correctness of the transformation. To address this issue, Danvy and Filinski (1992, Figure 2) propose a "one-pass" CPS transformation, which is so called because it produces no administrative redexes and therefore does not require a second pass during which such redexes are eliminated. Furthermore, they give a "properly tail-recursive" variant of the transformation (Danvy and Filinski, 1992, Figure 3), where care is taken not to produce a harmful η-redex when a tail call is translated. Finally, they establish the correctness of this transformation via a simple simulation argument. If the source program is able to make one step of computation, then the transformed program is able to follow suit in one or more steps (Danvy and Filinski, 1992, Lemma 3).

Danvy and Filinski's transformation is presented in a "higher-order" form, where some of the transformation functions are parameterized with "transformation-time continuations"

F. Pottier
Inria Paris E-mail: Francois.Pottier@inria.fr

which are transformation-time functions of terms to terms. Danvy and Nielsen (2003) later propose a simpler but less elegant "first-order" formulation, where transformation-time continuations are just terms. When the source language is the pure λ-calculus, the two formulations define the same transformation, so the same simulation property holds (Danvy and Nielsen, 2003, Lemma 2). When the source language is extended with a let construct, the two transformations no longer coincide: Danvy and Nielsen's transformation produces some administrative redexes which Danvy and Filinski's transformation eliminates (§6).

In this paper, we set out to give a crisp, machine-checked account of Danvy and Filinski's properly tail-recursive CPS transformation, in the setting of the pure λ-calculus extended with a let construct. Although it may seem as if every aspect of the CPS transformation has been documented already in the literature, we encounter a few unexpected difficulties along the way, and make the following contributions:

1. We make the (retrospectively entirely obvious) observation that a transformation-time continuation can be represented as a context, that is, as a term where a distinguished bound variable serves as a named hole. This leads us to propose a first-order formulation of the CPS transformation that is very close to Danvy and Filinski's higher-order formulation and does not exhibit the shortcomings of Danvy and Nielsen's first-order formulation.
2. We reduce the duplication inherent in Danvy and Filinski's formulation. Instead of two transformation functions, which are respectively used in "tail" and "nontail" contexts, we define a single transformation function, which receives information about the context as part of its continuation argument.
3. We represent variables as de Bruijn indices and show that, given the current state of the art, this representation does not pose a significant obstacle in the definition of the CPS transformation or in its correctness proof. We rely on Autosubst (Schäfer et al, 2015) to eliminate boilerplate definitions and to automate many low-level proof obligations about substitutions. Thus, we improve on some of the prior work, such as Minamide and Okuma's (2003), who use traditional named variables and have to explicitly account for α-equivalence and freshness, and Dargaye and Leroy's (2007), who use de Bruijn indices, but go through a nonstandard intermediate language, equipped with two name spaces. We emphasize that a "lifting" operation—that is, the application of an injective renaming—should be read as an "end-of-scope" operator, which indicates that a certain set of variables go out of scope. We present an interesting statement (Lemma 3) that is universally quantified in such a renaming θ.
4. Much to our surprise, we find that Danvy and Filinski's simulation argument (1992, Lemma 3) breaks down in the presence of let constructs. Indeed, although Danvy and Filinski (1992, Figure 4) extend the CPS transformation to deal with let constructs, among other features, they do not extend its correctness proof. We provide a counterexample and repair the proof by proposing a novel, slightly more complex simulation diagram, which involves call-by-value parallel reduction. The theory of parallel reduction (Takahashi, 1995, Crary, 2009) is used to conclude the proof.
5. Although Danvy and Filinski (1992) remark informally that "the transformation always terminates (and in essentially linear time)", we find that, when variables are represented as de Bruijn indices, it seems nontrivial to formulate the CPS transformation so that it runs efficiently. To address this problem, we propose a novel higher-order formulation of the transformation, which avoids all "lifting" operations by relying on "relocatable" terms and continuations. We prove it equivalent to our earlier formulation, and (informally and experimentally) check that its time complexity is $O(n \log n)$.

$$\begin{aligned}
\mathit{apply}\ (\mathsf{o}\ k)\ v &= k \mathbin{@} v && \text{– an object-level application}\\
\mathit{apply}\ (\mathsf{m}\ \kappa)\ v &= \kappa\ v && \text{– a metalevel application}\\
\mathit{reify}\ (\mathsf{o}\ k) &= k && \text{– a no-op}\\
\mathit{reify}\ (\mathsf{m}\ \kappa) &= \lambda x.(\kappa\ x) \qquad \text{where } x \,\#\, \kappa && \text{– a two-level } \eta\text{-expansion}
\end{aligned}$$

Fig. 1 Meta-level operations on continuations — informal version

$$\begin{array}{lrcll}
1. & (\!|x|\!) & = & x & \\
2. & (\!|\lambda x.t|\!) & = & \lambda x.\lambda y.\llbracket t\rrbracket\{\,\mathsf{o}\ y\,\} & \text{where } y \,\#\, t\\
3. & \llbracket v\rrbracket\{\,c\,\} & = & \mathit{apply}\ c\ (\!|v|\!) & \\
4. & \llbracket t_1 \mathbin{@} t_2\rrbracket\{\,c\,\} & = & \llbracket t_1\rrbracket\{\,\mathsf{m}\ v_1 \Rightarrow \llbracket t_2\rrbracket\{\,\mathsf{m}\ v_2 \Rightarrow v_1 \mathbin{@} v_2 \mathbin{@} (\mathit{reify}\ c)\,\}\,\} & \\
5. & \llbracket \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2\rrbracket\{\,c\,\} & = & \llbracket t_1\rrbracket\{\,\mathsf{m}\ v_1 \Rightarrow \mathsf{let}\ x = v_1\ \mathsf{in}\ \llbracket t_2\rrbracket\{\,c\,\}\,\} & \text{where } x \,\#\, c
\end{array}$$

Fig. 2 A higher-order, one-pass, call-by-value CPS transformation — informal version

Our definitions and proofs have been machine-checked using Coq and are electronically available (Pottier, 2017).

The paper is laid out as follows. We present the CPS transformation, first in a traditional pencil-and-paper style (§2), then in a formal style (§3), where variables are represented as de Bruijn indices. We state three fundamental lemmas on the interaction between the transformation and substitutions (§4), then prove that the transformation is semantics-preserving (§5) and discuss how the presence of let constructs breaks Danvy and Filinski's simulation diagram (§6). Finally, we present an efficient formulation of the transformation (§7), which we prove correct, before reviewing the related work (§8) and concluding (§9).

2 A one-pass call-by-value CPS transformation — informal version

We consider a core λ-calculus whose terms are $t ::= x \mid \lambda x.t \mid t \mathbin{@} t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t$. A value v (also, k) is a term of the form x or $\lambda x.t$. We wish to define two metalevel functions, namely:

$(\!\lvert v\rvert\!)$	the CPS transformation of the value v
$\llbracket t\rrbracket\{\,c\,\}$	the CPS transformation of the term t with continuation c

The latter form can be read informally as "with the result of evaluating t, do c". We let a continuation c be either a term k, also known as an object-level continuation, or a metalevel function κ of terms to terms, also known as a metalevel continuation.[1] The purpose of this distinction is to avoid the construction of so-called administrative redexes (Danvy and Filinski, 1992, §1). We use explicit injections and write $c ::= \mathsf{o}\ k \mid \mathsf{m}\ \kappa$. The injections $\mathsf{o}\ \cdot$ and $\mathsf{m}\ \cdot$, although noisy, are necessary (even on paper) to avoid ambiguity.

Throughout the paper, we write $v \Rightarrow t$ for a metalevel abstraction ("the function that maps v to t") and $\kappa\, v$ for a metalevel application ("the result of applying κ to v"). A metalevel function $v \Rightarrow t$ can be intuitively thought of as a term t with a hole named v. A metalevel

[1] We closely follow Danvy and Filinski's higher-order one-pass formulation (1992, Figure 3). However, whereas Danvy and Filinski define two functions $\llbracket\cdot\rrbracket'$ and $\llbracket\cdot\rrbracket$, which respectively expect a term-level continuation and a metalevel continuation, we define a single function $\llbracket\cdot\rrbracket\{\,\cdot\,\}$, whose second argument is either an object-level continuation or a metalevel continuation. This eliminates a significant amount of duplication in the definition. The existence of such a formulation was known to Danvy and Filinski (1992, §2.6), who write that "[one could] instrument the translation with an inherited attribute identifying tail-call contexts".

$$
\begin{array}{rcll}
\mathit{apply} & : & \mathsf{cont} \to \mathsf{term} \to \mathsf{term} & \\
\mathit{apply}\ (\mathsf{o}\ k)\ v & = & k \ @\ v & \text{– an object-level application} \\
\mathit{apply}\ (\mathsf{m}\ \kappa)\ v & = & \kappa[v/] & \text{– a metalevel substitution operation} \\
\\
\mathit{reify} & : & \mathsf{cont} \to \mathsf{term} & \\
\mathit{reify}\ (\mathsf{o}\ k) & = & k & \text{– a no-op} \\
\mathit{reify}\ (\mathsf{m}\ \kappa) & = & \lambda\, \kappa & \text{– a two-level } \eta\text{-expansion} \\
\\
\cdot[\sigma] & : & \mathsf{cont} \to \mathsf{cont} & \\
(\mathsf{o}\ k)[\sigma] & = & \mathsf{o}\ (k[\sigma]) & \text{– apply } \sigma \\
(\mathsf{m}\ \kappa)[\sigma] & = & \mathsf{m}\ (\kappa[\Uparrow \sigma]) & \text{– apply } \sigma \text{ under the binding construct } \mathsf{m}
\end{array}
$$

Fig. 3 Meta-level operations on continuations — formal version

application $t\ v$ can be thought of as the term obtained by filling the hole in t with the value v. In §7, we write *let* $v = t_1$ *in* t_2 for a metalevel local definition.

Two key operations on continuations are *apply* and *reify* (Figure 1). The term *apply* $c\ v$ can be thought of as the application of the continuation c to the value v. This is either an object-level application or a metalevel application, depending on the nature of c. The term *reify* c can be thought of as the continuation c, represented as a term. If c is an object-level continuation $\mathsf{o}\ k$, then *reify* c is just k. If c is a metalevel continuation $\mathsf{m}\ \kappa$, then *reify* c is $\lambda x.(\kappa\ x)$, that is, a λ-abstraction whose formal parameter is a fresh variable x and whose body is obtained by applying κ to the term x.

The call-by-value CPS transformation is defined in Figure 2. The functions $(\!|\cdot|\!)$ and $[\![\cdot]\!]\,\{\,\cdot\,\}$ are defined in a mutually inductive manner.

Equations 1 and 2 define the translation $(\!|v|\!)$ of a value v. A variable is translated to itself, while a function of one parameter x is translated to a function of two parameters[2] x and y, where the fresh variable y stands for a continuation. The function body, t, is translated with object-level continuation y.

The remaining equations define the translation $[\![t]\!]\,\{\,c\,\}$ of a term t under a continuation c. Equation 3 states that if the term t happens to be a value v, then its translation $[\![v]\!]\,\{\,c\,\}$ is the application of the continuation c to the value $(\!|v|\!)$. Equation 4 defines the term $[\![t_1\ @\ t_2]\!]\,\{\,c\,\}$ so that the translation of t_1 runs first, yielding a value denoted by the metavariable v_1. Then, the translation of t_2 is executed, yielding a value denoted by v_2. Finally, an object-level application of v_1 to the two arguments v_2 and *reify* c is built, in accordance with the fact that a translated function takes two arguments: a value and a continuation. In equation 5, the translation of the term t_1 is executed first. Its value, denoted by v_1, is bound via a let construct to the variable x. The continuation c is used, unchanged, in the translation of t_2, reflecting the fact that t_2 occurs in tail position in the term $\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$.

3 A one-pass call-by-value CPS transformation — formal version

The informal definitions in the previous section (§2) rely on the nominal representation of λ-terms, which is standard in pencil-and-paper proofs. Although this representation has well-understood foundations (Pitts, 2005) and is natively supported by certain proof assistants, such as Nominal Isabelle (Urban, 2008, Huffman and Urban, 2010), it does not seem very convenient for use in Coq as of today. In the formal part of the paper, we switch to de Bruijn's representation (1972), which is well supported in Coq, thanks to the Autosubst

[2] Ideally, the target language of the translation would have functions of arity 2. For simplicity, we prefer to unify the source and target languages, and represent a function of arity 2 as a curried function.

$$
\begin{array}{lrcl}
 & (\!|\cdot|\!) & : & \text{term} \to \text{term} \\
1. & (\!|x|\!) & = & x \\
2. & (\!|\lambda t|\!) & = & \lambda\lambda(\llbracket\uparrow^1 t\rrbracket\,\{\,\mathsf{o}\,0\,\}) \\
 & & & \\
 & \llbracket\cdot\rrbracket\,\{\,\cdot\,\} & : & \text{term} \to \text{cont} \to \text{term} \\
3. & \llbracket v\rrbracket\,\{\,c\,\} & = & \mathit{apply}\; c\; (\!|v|\!) \\
4. & \llbracket t_1 \;@\; t_2\rrbracket\,\{\,c\,\} & = & \llbracket t_1\rrbracket\,\{\,\mathsf{m}\;\llbracket\uparrow^1 t_2\rrbracket\,\{\,\mathsf{m}\;1 \;@\; 0 \;@\; \uparrow^2(\mathit{reify}\; c)\,\}\,\} \\
5. & \llbracket \mathsf{let}\; t_1 \;\mathsf{in}\; t_2\rrbracket\,\{\,c\,\} & = & \llbracket t_1\rrbracket\,\{\,\mathsf{m}\;\mathsf{let}\;0\;\mathsf{in}\;\llbracket\uparrow^1_1 t_2\rrbracket\,\{\,\uparrow^2 c\,\}\,\}
\end{array}
$$

Fig. 4 A first-order, one-pass, call-by-value CPS transformation — formal version

library (Schäfer et al, 2015). Autosubst offers a simplification tactic, `asimpl`, which attempts to bring substitutions (and applications of a substitution to a term) into a canonical form. This greatly reduces (yet does not quite eliminate) the burden of reasoning about renamings and substitutions.

Furthermore, instead of a higher-order formulation, where terms with named holes are represented by metalevel functions, we switch to a first-order formulation, where terms with named holes are represented as ordinary terms, and ordinary variables serve as holes. Indeed, the former representation, a form of higher-order abstract syntax (Pfenning and Elliott, 1988), includes undesired "exotic terms", that is, metalevel functions that inspect their argument, instead of just building on top of it. Working with that representation would require defining a notion of "well-behaved" metalevel function and proving that every metalevel function of interest is well-behaved, which would be rather tedious. While there are variants of higher-order abstract syntax that rule out exotic terms, such as parametric higher-order abstract syntax (Chlipala, 2008), we prefer to stick with a first-order representation, where a "named hole" is just a variable, and a variable is just a de Bruijn index.

Interestingly, the higher-order style re-appears further on in the paper (§7), where it is used in an efficient implementation of the CPS transformation.

3.1 Notation

From here on, a **variable** x is a natural number, and we work with a λ-calculus whose **terms** are $t ::= x \mid \lambda t \mid t \;@\; t \mid \mathsf{let}\; t \;\mathsf{in}\; t$. We write "term" for the type of terms. A **value** v (also, k) is a term of the form x or λt.

A **substitution** σ is a total function of variables to terms. A **value substitution** is a substitution σ such that, for every variable x, the term $\sigma(x)$ is a value. A **renaming** is a substitution σ such that, for every variable x, the term $\sigma(x)$ is a variable. (A renaming is not necessarily injective.) We write $+i$ for the renaming $x \Rightarrow x+i$. We write $v \cdot \sigma$ for the substitution that maps 0 to v and $1+x$ to $\sigma(x)$. We write $v/$ for the substitution $v \cdot id$. We write $\Uparrow\sigma$ for the substitution that maps 0 to 0 and that maps $1+x$ to $1+\sigma(x)$, for every x. We write $t[\sigma]$ for the (capture-avoiding) application of the substitution σ to the term t. This operation is defined as follows:

$$
\begin{array}{rclrcl}
x[\sigma] & = & \sigma(x) & (\lambda t)[\sigma] & = & \lambda(t[\Uparrow\sigma]) \\
(t_1 \;@\; t_2)[\sigma] & = & (t_1[\sigma]) \;@\; (t_2[\sigma]) & (\mathsf{let}\; t_1 \;\mathsf{in}\; t_2)[\sigma] & = & \mathsf{let}\;(t_1[\sigma])\;\mathsf{in}\;(t_2[\Uparrow\sigma])
\end{array}
$$

In Coq, this definition is automatically generated by Autosubst, based on programmer-supplied indications of where variables are bound. We write $\sigma_1\,;\sigma_2$ for the composition of σ_1 and σ_2, that is, for the substitution that maps x to $x[\sigma_1][\sigma_2]$.

We write $\uparrow^i t$ for $t[+i]$. In short, $\uparrow^i t$ is the term obtained by adding i to every variable that appears free in the term t. The symbol $\uparrow^i$ can be intuitively read as an end-of-scope mark: it means that the i most-recently-introduced variables are *not* in scope in the term that follows (Bird and Paterson, 1999, Hendriks and van Oostrom, 2003).

We write $\uparrow^1_1 t$ for $t[\Uparrow(+1)]$. This renaming lifts every variable except 0 up by one, and can be understood as an end-of-scope mark for the variable 1.

3.2 The CPS transformation

A **continuation** c is either a value k, also known as an object-level continuation, or a term-with-a-hole κ, also known as a metalevel continuation. We write $c ::= \mathsf{o}\,k \mid \mathsf{m}\,\kappa$, and write "cont" for the type of continuations. A term-with-a-hole is just a term, where, by convention, the variable 0 represents the hole (and, accordingly, every other variable is lifted up by 1). The injection m can be viewed as a binding construct, just like λ: indeed, it introduces a new variable, numbered 0.

Three key operations on continuations are *apply*, *reify*, and substitution (Figure 3). The definitions of *apply* and *reify* are analogous to those presented earlier (Figure 1). In the second line of *apply*'s definition, the operation of filling the hole in κ with a value v is now just a substitution $\kappa[v/]$. In the second line of *reify*'s definition, the variable 0, which is bound by m on the left-hand side, becomes bound by λ on the right-hand side. The operation of applying a substitution σ to a continuation c is defined in the obvious way. It is used in Figure 4, as $\uparrow^2 c$ is sugar for $c[+2]$. It is also used in the statement of several fundamental lemmas (§4).

The definition of the CPS transformation appears in Figure 4. In Equation 2, instead of a freshness side condition $y \# t$ (Figure 2), an end-of-scope mark is used: the term t becomes $\uparrow^1 t$ when it is brought down into the scope of the second λ. Similarly, in Equation 4, the term t_2 is preceded with $\uparrow^1$ because it is brought down into the scope of one m binder, and the term *reify* c is preceded with $\uparrow^2$ because it is brought down into the scope of two m binders. The two "named holes" denoted by v_1 and v_2 in Figure 2 are now represented by the variables 1 and 0, respectively. In Equation 5, similarly, the "named hole" v_1 is now represented by the variable 0. The term t_2 is preceded with $\uparrow^1_1$ because it is brought down into the scope of an m binder that binds variable 1. We must therefore indicate that this variable is not in scope in t_2.

Because a continuation $c ::= \mathsf{o}\,k \mid \mathsf{m}\,\kappa$ is a syntactic object, our formulation of the CPS transformation is first-order. It nevertheless closely resembles Danvy and Filinski's higher-order formulation (1992, Figure 3). The two are in fact extensionally equal: they are two formulations of the same transformation. Our formulation does not share the shortcomings of Danvy and Nielsen's first-order formulation (2003, Figure 2), to wit: (1) the transformation of applications requires distinguishing 4 cases, and the transformation of n-tuples would require distinguishing 2^n cases; and (2) when the source language is extended with let constructs, Danvy and Nielsen's transformation produces certain administrative redexes which Danvy and Filinski's transformation eliminates (§6).

It should be noted that the definition in Figure 4 is not directly accepted by Coq as an inductive definition. This is due in part to Coq's poor support for mutually inductive definitions: the recursive call $(\!|v|\!)$ in Equation 3 involves v, whereas Coq would insist that it should involve a subterm of v. A deeper reason is that the recursive call $[\![\uparrow^1 t]\!]\{\dots\}$ in Equation 2 involves $\uparrow^1 t$, which is not a subterm of λt. We work around the former problem by first defining $[\![t]\!]\{c\}$ (inlining away $(\!|v|\!)$, which does not cause much duplication), then recover

$(\!|v|\!)$ as $[\![v]\!]\,\{\, done \,\}$, where *done* is defined as $\mathsf{m}\ 0$. This trick is found in Danvy and Filinski's paper (1992, Definition 1), where $(\!|v|\!)$ is written $\Psi(v)$. We work around the latter problem by using well-founded recursion over the size of terms, in a style explained by Chlipala (2013, Chapter 7). These workarounds are rather irritating, as they represent a barrier to entry, especially for students. Although these issues are well-known (Barthe et al, 2006), better support in Coq for general recursive functions still seems to be lacking. Fortunately, the difficulties encountered in the definition of the CPS transformation do not affect reasoning about this transformation. We prove that the two transformation functions satisfy the equations of Figure 4, and set up a tailor-made induction principle so that statements such as Lemmas 1-3 can be proved by mutual induction over the size of terms, without fuss.

4 Basic lemmas

We now state three basic lemmas that describe how the CPS transformation interacts with renamings and substitutions. The three statements have similar structure. All three of them are proved in the same manner, namely by mutual induction over the size of v and t, using the mutual induction principle mentioned above. The proofs are not difficult: for each lemma, a proof script of about 25 nonblank, noncomment lines is required.

The CPS transformation commutes with renamings: in short, a renaming σ can be pushed down into both sides of $[\![t]\!]\,\{\, c \,\}$.

Lemma 1 (Renaming) *For every renaming σ,*

1. *for every value v,* $(\!|v|\!)[\sigma] = (\!|v[\sigma]|\!)$.
2. *for every term t and continuation c,* $([\![t]\!]\,\{\, c \,\})[\sigma] = [\![t[\sigma]]\!]\,\{\, c[\sigma] \,\}$.

As an almost-immediate corollary, we obtain the following equality:

$$\Uparrow(\sigma\,;(\!|\cdot|\!)) = (\Uparrow\sigma)\,;(\!|\cdot|\!)$$

That is, if a substitution σ' can be expressed as the composition $\sigma\,;(\!|\cdot|\!)$ of a substitution σ and the CPS transformation, then $\Uparrow\sigma'$ can be similarly expressed as the composition $(\Uparrow\sigma)\,;(\!|\cdot|\!)$. This equality is exploited in the proof of the substitution lemma, which follows. This seems to be the reason why we must establish Lemma 1 before Lemma 2, even though Lemma 2 subsumes Lemma 1.

The CPS transformation commutes with substitutions σ' that can be expressed as a composition $\sigma\,;(\!|\cdot|\!)$, where σ is a value substitution. In short, such a substitution σ' can be pushed down into both sides of $[\![t]\!]\,\{\, c \,\}$, becoming σ on the term side and remaining σ' on the continuation side. This reflects the fact that t represents a source term, which is subject to the transformation, whereas c represents a target term, which is not subject to it. This result corresponds to Danvy and Filinski's Lemma 2 (1992) and to the first part of Danvy and Nielsen's Lemma 1 (2003).

Lemma 2 (Substitution) *For every value substitution σ, for every substitution σ', where σ' is equal to $\sigma\,;(\!|\cdot|\!)$,*

1. *for every value v,* $(\!|v|\!)[\sigma'] = (\!|v[\sigma]|\!)$.
2. *for every term t and continuation c,* $([\![t]\!]\,\{\, c \,\})[\sigma'] = [\![t[\sigma]]\!]\,\{\, c[\sigma'] \,\}$.

The third and last fundamental lemma involves "kubstitutions", a word that we coin for substitutions that affect the continuation, but not the term. We wish to express the informal idea that, "if σ does not affect the term t, then σ can be pushed down into $[\![t]\!]\,\{\,c\,\}$, where it vanishes on the term side, and remains σ on the continuation side". This is an interesting statement, whose standard (informal, pencil-and-paper) formulation is simple, yet whose formulation in de Bruijn style requires a little thought. Indeed, the condition that "σ does not affect the term t" should *not* be expressed by the equation $t[\sigma] = t$. That would be too restrictive. For instance, the substitution $v/$ intuitively "does not affect" the term $\uparrow^1 t$, where the variable 0 does not occur free; yet, the result of the substitution application $(\uparrow^1 t)[v/]$ is t, which is not equal to $\uparrow^1 t$. In other words, a substitution that "does not affect" a term can still cause its free variables to be renumbered.

A more general approach is to allow the term t to carry an end-of-scope mark, which is represented by an injective renaming θ. (We have already encountered several renamings that can be interpreted as end-of-scope marks, such as $\uparrow^1$, $\uparrow^2$, and $\uparrow^1_1$.) One then requires the substitution σ to act only upon the variables which θ causes to go out of scope: this is very elegantly expressed by the equation $\theta\,;\sigma = id$. (This equation implies that θ is an injective renaming.) For instance, instantiating θ with $\uparrow^1$ and σ with $v/$ satisfies this equation.

We thus obtain the following statement. In short, if the composition $\theta;\sigma$ vanishes, then σ can be pushed down into $[\![t[\theta]]\!]\,\{\,c\,\}$, where it annihilates with θ on the term side, and remains σ on the continuation side.

Lemma 3 (Kubstitution) *For all substitutions θ and σ, where the composition $\theta\,;\sigma$ is the identity substitution,*

1. *for every value v, $(\!|v[\theta]|\!)[\sigma] = (\!|v|\!)$.*
2. *for every term t and continuation c, $[\![(t[\theta]]\!]\,\{\,c\,\})[\sigma] = [\![t]\!]\,\{\,c[\sigma]\,\}$.*

It might seem as if Lemma 3 is a corollary of Lemma 2. In fact, it is not, as it does not require σ to be expressible under the form $_\,;(\!|\cdot|\!)$.

As a corollary, we get the equality:

$$([\![\uparrow^1 t]\!]\,\{\,c\,\})[v/] = [\![t]\!]\,\{\,c[v/]\,\}$$

This equality is used without justification by Danvy and Filinski (1992, proof of Lemma 3) and forms the second part of Danvy and Nielsen's Lemma 1 (2003). The statement found there may seem simpler, but in reality lacks a freshness hypothesis, which corresponds to our use of $\uparrow^1$.

Using Lemma 3, we establish a few equations that clarify how the CPS transformation behaves when applied to a term whose immediate subterms are values.

Lemma 4 (Special cases) *For all values v_1, v_2, for every term t_2, for every continuation c,*

$$\begin{aligned}
[\![v_1 \mathbin{@} t_2]\!]\,\{\,c\,\} &= [\![t_2]\!]\,\{\,\mathsf{m}\uparrow^1 (\!|v_1|\!) \mathbin{@} 0 \mathbin{@} \uparrow^1 (\mathit{reify}\ c)\,\}\\
[\![v_1 \mathbin{@} v_2]\!]\,\{\,c\,\} &= (\!|v_1|\!) \mathbin{@} (\!|v_2|\!) \mathbin{@} (\mathit{reify}\ c)\\
[\![\mathsf{let}\ v_1\ \mathsf{in}\ t_2]\!]\,\{\,c\,\} &= \mathsf{let}\ (\!|v_1|\!)\ \mathsf{in}\ [\![t_2]\!]\,\{\uparrow^1 c\,\}
\end{aligned}$$

5 Correctness of the CPS transformation

In an untyped setting, a term must exhibit one of three behaviors: it either converges (that is, reduces in zero, one, or more steps to a value), or diverges (that is, admits an infinite

reduction sequence), or reduces to a stuck term (a term that cannot reduce, yet is not a value). We now prove that the CPS transformation is correct: that is, if a source term t exhibits one of these behaviors, then the transformed term $[\![t]\!]\,\{\,done\,\}$ exhibits the same behavior, where *done* is the empty-context continuation, $\mathsf{m}\ 0$.

We write $\longrightarrow_{\mathrm{cbv}}$ for the (small-step) call-by-value reduction relation, whose well-known definition we omit. We write $\Longrightarrow_{\mathrm{cbv}}$ for the parallel call-by-value reduction relation, whose definition we also omit: the reader is referred to Crary's work (2009). Parallel reduction strictly contains reduction: $(\longrightarrow_{\mathrm{cbv}}) \subset (\Longrightarrow_{\mathrm{cbv}})$. In short, parallel reduction differs from reduction in that (1) it can contract several redexes at once and (2) it can contract redexes under an arbitrary context, including under a λ-abstraction and in the right-hand side of a let construct.

The lemmas in this section are simple and have relatively short proofs (the proof scripts for all lemmas, together, take up about 60 nonblank, noncomment lines). They may seem quite a bit more involved than Danvy and Filinski's simulation statement (1992, Lemma 3), also found in Danvy and Nielsen's paper (2003, Lemma 2). Indeed, the presence of the let construct (which the papers just cited do not handle) complicates matters, requiring us to define a notion of similarity between continuations, to prove several preliminary lemmas, and to establish a simulation statement that involves parallel reduction (Lemma 7). In the next section (§6), we justify more precisely why this added complexity seems unavoidable.

Definition 1 (Similarity of continuations) The assertion "c_1 is **similar** to c_2" is inductively defined by the following two rules:

1. $\mathsf{o}\ (reify\ c)$ is similar to c.
2. If $\kappa_1 \Longrightarrow_{\mathrm{cbv}} \kappa_2$ holds, then $\mathsf{m}\ \kappa_1$ is similar to $\mathsf{m}\ \kappa_2$.

The first rule in Definition 1 must be present because, in the proof of Lemma 7, in the case of β_v-reduction, we wish to apply Lemma 6, instantiated with $\mathsf{o}\ (reify\ c)$ and c. The second rule in Definition 1 must be present because, in the proof of Lemma 6, in the case where t is a let construct, we wish to apply the induction hypothesis, instantiated with two continuations $\mathsf{m}\ \kappa_1$ and $\mathsf{m}\ \kappa_2$ which we can prove (using the induction hypothesis, again) satisfy $\kappa_1 \Longrightarrow_{\mathrm{cbv}} \kappa_2$.

The following lemma is easily established by case analysis. The proof of its second item relies on the fact that parallel reduction under a λ-abstraction is permitted.

Lemma 5 (Application and reification of similar continuations) *For all continuations c_1 and c_2, if c_1 is similar to c_2, then:*

1. *for every value v, apply $c_1\ (\!|v|\!) \Longrightarrow_{cbv}$ apply $c_2\ (\!|v|\!)$ holds;*
2. *reify $c_1 \Longrightarrow_{cbv}$ reify c_2 holds.*

The next lemma is easily established by induction over the size of the term t. The proof of the case where t is a let construct relies on the fact that parallel reduction in the right-hand side of a let construct is permitted.

Lemma 6 (Reduction in the continuation) *For all continuations c_1 and c_2, if c_1 is similar to c_2, then, for every term t, $[\![t]\!]\,\{\,c_1\,\} \Longrightarrow_{cbv} [\![t]\!]\,\{\,c_2\,\}$ holds.*

We then reach the main simulation diagram. This is a forward simulation diagram, which states that the transformed term is able to simulate every step of computation taken by the source term. In contrast with the statement proved by Danvy and Filinski (1992, Lemma 3) and Danvy and Nielsen (2003, Lemma 2), we must allow the transformed term to take not just one or more reduction steps $\longrightarrow^{+}_{\mathrm{cbv}}$, but also one parallel reduction step $\Longrightarrow_{\mathrm{cbv}}$. As parallel reduction is a reflexive relation, this last step can be trivial.

Lemma 7 (Simulation) *For all terms t_1 and t_2, for every continuation c, provided reify c is a value, $t_1 \longrightarrow_{cbv} t_2$ implies $[\![t_1]\!]\,\{c\}\ (\longrightarrow^{+}_{cbv} \cdot \Longrightarrow_{cbv})\ [\![t_2]\!]\,\{c\}$.*

From this result, there follows that the CPS transformation (initiated with the identity continuation) is correct: that is, it preserves convergence to a value, divergence, and "going wrong", that is, reducing to a stuck term.

Lemma 8 (Correctness) *For every term t,*

1. *if $t \longrightarrow^{\star}_{cbv} v$ holds, where v is a value,
then there exists a value v' such that $[\![t]\!]\,\{done\} \longrightarrow^{\star}_{cbv} v'$ and $v' \Longrightarrow^{\star}_{cbv} (\!|v|\!)$.*
2. *if $t \longrightarrow^{\infty}_{cbv}$ holds,
then $[\![t]\!]\,\{done\} \longrightarrow^{\infty}_{cbv}$ holds as well.*
3. *if $t \longrightarrow^{\star}_{cbv} t'$ holds, where t' is stuck,
then $[\![t]\!]\,\{done\} \longrightarrow^{\star}_{cbv} t''$, where t'' is stuck.*

The proof of the second item above relies on the fact that reduction and parallel reduction commute, that is, $\Longrightarrow^{\star}_{\text{cbv}} \cdot \longrightarrow^{+}_{\text{cbv}}$ is a subset of $\longrightarrow^{+}_{\text{cbv}} \cdot \Longrightarrow^{\star}_{\text{cbv}}$. This fact is proved by Crary (2009), based on Takahashi's results for call-by-name λ-calculus (Takahashi, 1995). We port Crary's results to Coq, so as to offer a self-contained proof of our claims. The proofs of the first and third item also exploit the fact that reduction and parallel reduction commute, but require a more precise statement of this fact, namely, Crary's Bifurcation lemma (2009, Lemma 9).

It is easy to prove that the behavior of a CPS-transformed term is the same under call-by-value and call-by-name evaluation. Indeed, to establish this result, it suffices to remark that, in such a term, the right-hand side of every application must be a value, and the left-hand side of every let construct must be a value. This property (which is preserved by reduction) is sufficient to guarantee indifference.

Lemma 9 (Indifference) *For every term t, the term $[\![t]\!]\,\{done\}$ exhibits the same reduction sequence under call-by-value and call-by-name reduction semantics.*

6 How let constructs complicate matters

Simplifications in the absence of let. In the absence of a let construct, the statements and proofs in the previous section can be simplified as follows. The second rule in the definition of similarity (Definition 1) can be removed. In the first item of Lemma 5, parallel reduction $\Longrightarrow_{\text{cbv}}$ can be replaced with at most one step of reduction, $\longrightarrow^{?}_{\text{cbv}}$. In the second item of Lemma 5, parallel reduction can (almost miraculously) be replaced with an equality: indeed, *reify* $(\mathsf{o}\,(\textit{reify}\ c)) = \textit{reify}\ c$ holds. In the statement of Lemma 6, parallel reduction can then be replaced with $\longrightarrow^{?}_{\text{cbv}}$. Finally, in the statement of Lemma 7, the relation $(\longrightarrow^{+}_{\text{cbv}} \cdot \Longrightarrow_{\text{cbv}})$ can be replaced with just $\longrightarrow^{+}_{\text{cbv}}$, yielding the simple simulation diagram found in the papers by Danvy and Filinski (1992, Lemma 3) and Danvy and Nielsen (2003, Lemma 2). Parallel reduction is not needed any more, so that correctness (Lemma 8) can be established without appeal to the theory of parallel reduction.

With let, *Danvy and Filinski's transformation violates the simple simulation diagram.* Does the simulation diagram necessarily become more complex in the presence of let? The answer is positive. In the presence of let, the CPS transformation that we study (§3), which coincides

with Danvy and Filinski's transformation, does not satisfy the simple simulation diagram. Here is an example that demonstrates this. Let t_1 stand for the term $(\lambda(\mathsf{let}\ 0\ \mathsf{in}\ 0))\ @\ (\lambda 0)$, which in informal syntax would be written:

$$(\lambda z.\mathsf{let}\ w = z\ \mathsf{in}\ w)\ @\ (\lambda x.x)$$

This term reduces to t_2, which in de Bruijn's notation is $\mathsf{let}\ \lambda 0\ \mathsf{in}\ 0$ and in informal syntax would be written:

$$\mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ w$$

According to the simple simulation diagram, $[\![t_1]\!]\,\{\,done\,\} \longrightarrow^{+}_{\mathsf{cbv}} [\![t_2]\!]\,\{\,done\,\}$ should hold. Yet, this property is false: the term $[\![t_1]\!]\,\{\,done\,\}$ does not reduce (in any number of call-by-value reduction steps) to the term $[\![t_2]\!]\,\{\,done\,\}$. (We have checked this in Coq.) Let us explain why that is the case. The term $[\![t_1]\!]\,\{\,done\,\}$, in informal notation, is:

$$(\lambda z.\lambda k.\mathsf{let}\ w = z\ \mathsf{in}\ k\ @\ w)\ @\ (\lambda x.\lambda k.\ k\ @\ x)\ @\ (\lambda w.w)$$

This term reduces in two β_v steps to:

$$\mathsf{let}\ w = \lambda x.\lambda k.\ k\ @\ x\ \mathsf{in}\ (\lambda w.w)\ @\ w$$

Unfortunately, the term $[\![t_2]\!]\,\{\,done\,\}$, in informal notation, is:

$$\mathsf{let}\ w = \lambda x.\lambda k.\ k\ @\ x\ \mathsf{in}\ w$$

We have a "near miss". The last two displayed terms differ by the contraction of a β_v-redex, which takes place in the right-hand side of a let construct. Such a contraction is not permitted by the relation $\longrightarrow_{\mathsf{cbv}}$.

This counter-example is one of two counter-examples of minimal size (not counting variables, the term t_1 has size 4) and involves only one let construct, so it is arguably the "simplest" possible counter-example. It was found by an exhaustive enumeration procedure, illustrating the fact that testing can help disprove conjectures.

With let*, Danvy and Nielsen's transformation obeys the simple simulation diagram.* Danvy and Nielsen's transformation is extended with support for let constructs by Minamide and Okuma (2003, §4.4), who report that this extended transformation still obeys a simple simulation diagram: when the source program makes one step, the transformed program follows suit in zero or more steps (up to α-equivalence, which in Minamide and Okuma's paper is explicit). They formally verify this fact using Isabelle/HOL.

This does not contradict our findings, because Minamide and Okuma's transformation does not coincide with Danvy and Filinski's transformation, which we study. Indeed, the former produces administrative redexes which the latter eliminates. Consider, for instance, the term t_2' defined as $t_2\ @\ 0$, where the term t_2 is as above. In informal notation, the term t_2' can be written:

$$(\mathsf{let}\ w = \lambda x.x\ \mathsf{in}\ w)\ @\ y$$

Minamide and Okuma's transformation, applied to t_2' and to the identity continuation $\lambda z.z$, yields:

$$\mathsf{let}\ w = \lambda x.\lambda k.\ k\ @\ x\ \mathsf{in}\ (\lambda w.\ w\ @\ y\ @\ (\lambda z.z))\ @\ w$$

This term exhibits an administrative redex $(\lambda w.\dots)\ @\ w$. In contrast, Danvy and Filinski's transformation, applied to t_2' and to the object-level identity continuation $\circ\,\lambda z.z$, yields:

$$\mathsf{let}\ w = \lambda x.\lambda k.\ k\ @\ x\ \mathsf{in}\ w\ @\ y\ @\ (\lambda z.z)$$

This is no surprise: Danvy and Nielsen (2003, §7.2) note that their formulation of the CPS transformation cannot be extended with support for let in such a way that it "flattens nested blocks". They write: "We do not see how a first-order one-pass CPS transformation can flatten nested blocks in general if it is also to be compositional." We show that this is in fact possible: our formulation (§3) is first-order, compositional, and coincides with Danvy and Filinski's transformation. The downside of our formulation is that it is not efficient: we now turn to this issue.

7 An efficient implementation of the CPS transformation

Extracting OCaml code out of the Coq definitions of Figures 3 and 4 yields a working yet very inefficient implementation of the CPS transformation. Indeed, these definitions involve substitution under various guises. A substitution operation $\kappa[v/]$ is explicitly used in the definition of *apply*. The lifting operations $\uparrow^1$, $\uparrow^2$, and $\uparrow^1_1$ are also substitution operations. This gives rise to two problems. First, even if the operation of applying a substitution to a term was somehow efficiently implemented, its cost would still be at least linear in the size of the term, which implies that the time complexity of the CPS transformation would be at least quadratic. Second, Autosubst's implementation of this operation is inefficient. Indeed, for mathematical simplicity, Autosubst represents a substitution σ as a function. From an algorithmic point of view, this is not a good choice. The construction of the substitution $\Uparrow\sigma$, which takes place when a binder is entered, allocates a new closure. Thus, a substitution forms a linked list of closures in memory, whose length is at least the number of binders that have been entered. Therefore, the cost of applying a substitution to a variable is at least linear, and the cost of applying a substitution to a term is at least quadratic.

This naturally raises the questions: can the CPS transformation, viewed as a function of terms (in de Bruijn's representation) to terms (in the same representation), be efficiently implemented? Can this implementation be proved correct with respect to the definition that we have presented earlier (§3)? Because we found this "programming exercise" considerably more difficult than expected, we describe our solution.

In order to obtain an efficient implementation of the CPS transformation, it is necessary to avoid the use of lifting and substitution altogether. To avoid the substitution $\kappa[v/]$ of a value v into a term-with-a-hole κ, we revert to a higher-order formulation, where metalevel continuations are represented as metalevel functions. To avoid the lifting operations $\uparrow^1$, $\uparrow^2$, and $\uparrow^1_1$, we explicitly carry and maintain a renaming, which is applied on the fly. Of course, an efficient representation of renamings, which supports lookup and extension in logarithmic time, must be used.

7.1 Efficient de Bruijn renamings

To begin with, we need an efficient representation of renamings. Recall that a renaming is a total mapping of variables to variables, or more precisely, of de Bruijn indices to de Bruijn indices. This representation must efficiently support the following four operations:

1. Build the identity renaming *id*.
2. Given a renaming σ and a variable x, compute $\sigma(x)$.
3. Given a renaming σ, construct the renaming $\Uparrow\sigma$.
4. Given a renaming σ, construct the renaming $\sigma\,;(+1)$.

It should be obvious why the first two operations are needed. Operation 3 is required when a (λ-bound or let-bound) variable in the source program is translated to a (similarly λ-bound or let-bound) variable in the target program. Operation 4 is required when a variable that does not exist in the source program is introduced in the target program: this is the case of continuation variables.

To gain an intuition for these operations, it is useful to visualize a renaming σ as a bipartite graph on $\mathbb{N} \times \mathbb{N}$, where the source variables, labeled $0, 1, \ldots$, appear on the left-hand side; the target variables, labeled $0, 1, \ldots$, appear on the right-hand side; and where there is an edge from every source variable x to the target variable $\sigma(x)$. Operation 3 can then be viewed as renumbering every vertex on either side by adding 1 to its label; creating one new vertex, numbered 0, on either side; and adding a new edge between these new vertices. Visually, the original graph is untouched; two new vertices and a new edge are added to it. Operation 4 can be viewed as renumbering every vertex on the right-hand side by adding 1 to its label, and creating one new vertex, numbered 0, on the right-hand side. Visually, the original graph is again untouched; one new vertex is added to it.

It may not be obvious at first how these four operations can be efficiently implemented. A traditional finite map data structure, such as a balanced binary search tree, supports the insertion of new key-value pairs, but does not support renumbering every key in its domain or renumbering every value in its codomain. A binary random access list (Okasaki, 1999, Chapter 9) (Sozeau, 2007) supports the former operation, but not the latter.

Our solution to this puzzle is to internally represent a renaming using a finite map m of de Bruijn levels to de Bruijn levels. (de Bruijn levels are analogous to de Bruijn indices, but are counted in the opposite direction: a higher level denotes a more recent binder.) Let us assume, for a moment, that the height of the source name space is *src*, that is, only the source variables in the semi-open interval $[0, src)$ are of interest. Then, the conversion between a source de Bruijn index and a source de Bruijn level (in either direction) is performed by transforming x to $src - 1 - x$. Similarly, if the height of the target name space is *dst*, then the conversion between a target index and a target level is performed by transforming x' to $dst - 1 - x'$. Thus, if m is a map of levels to levels, and if the operation $find\ x\ m$ can be used to look up x in this map, then the corresponding transformation of indices to indices maps x to $dst - 1 - find\ (src - 1 - x)\ m$.

Formally speaking, let us assume that a correct and efficient implementation of finite maps is available. We assume that this data structure supports the operations *empty*, *add*, and *find*. (We axiomatize these operations in Coq and implement them in OCaml using OCaml's balanced binary search tree library.) Then, we represent a renaming as a triple (src, m, dst), where m is a finite map and *src* and *dst* are natural numbers. (Slightly cutting corners, we instruct Coq to represent natural numbers as OCaml machine integers.) The function *interpret* in Figure 5 defines how a triple (src, m, dst) should be interpreted as a renaming, that is, as a total function of variables to variables. The first equation in Figure 5 has been justified above. The second equation in Figure 5 causes the renaming to behave uniformly beyond the heights of the source and target name spaces: that is, the variables $src, src+1, \ldots$ are mapped to $dst, dst+1, \ldots$, and so on.

Definition 2 A triple (src, m, dst) **represents** a renaming σ if

1. m is a well-formed finite map;
2. the interval $[0 \ldots src)$ is contained in the domain of m;
3. the image of this interval through m is contained in the interval $[0 \ldots dst)$;
4. for every x, the equality $interpret\ (src, m, dst)\ x = \sigma(x)$ holds.

Lemma 10 *The four desired operations on renamings are implemented as follows:*

1. $(0, empty, 0)$ *represents the identity renaming.*
2. *If* (src, m, dst) *represents* σ*, then* $interpret\ (src, m, dst)\ x$ *is* $\sigma(x)$.
3. *If* (src, m, dst) *represents* σ*, then* $(src+1, add\ src\ dst\ m, dst+1)$ *represents* $\Uparrow \sigma$.
4. *If* (src, m, dst) *represents* σ*, then* $(src, m, dst+1)$ *represents* $\sigma\,;(+1)$.

7.2 Relocatable terms

In order to obviate the need for lifting operations, such as $\uparrow^1$, instead of constructing terms, whose free variables might later need to be renumbered, we construct relocatable terms. A **relocatable term** is a function, which, when applied to the height *dst* of the target name space, produces a term. Thus, the type "rterm" of relocatable terms is defined as nat $\rightarrow$ term.

In the following, we reuse the metavariables t, k, v to range over relocatable terms, as it would be cumbersome to have to invent a whole new set of metavariables. The type of a metavariable should always be clear from the context, as in the next definition, where t denotes a relocatable term and t' denotes a term. Similarly, in the next subsection (§7.3), the metavariable κ is reused to denote a metalevel function of relocatable terms to relocatable terms, and the metavariable c is reused to denote a relocatable continuation.

The meaning of a relocatable term is defined by a 3-place predicate whose parameters respectively have types rterm, term, and nat, as follows.

Definition 3 A relocatable term t **represents** a term t' **at time** *dst* if and only if, for every i, the application $t\ (dst+i)$ yields the term $\uparrow^i t'$.

This definition builds in the idea that the height of the target name space can only grow between the time *dst* when a relocatable term is constructed and the time $dst+i$ when this relocatable term is applied so as to obtain a term.

The following (trivial) lemma states that relocatable terms really are relocatable: that is, they do not need to be explicitly lifted. If the relocatable term t represents the term t' when the height of the target name space is *dst*, then the same relocatable term t automatically represents $\uparrow^i t'$ when the height of the target name space grows to $dst+i$.

Lemma 11 (Relocation) *If t represents t' at time dst, then t represents $\uparrow^i t'$ at time $dst+i$.*

We use the auxiliary functions in Figure 6 to construct relocatable terms.

The function call *var dst x* converts the de Bruijn index x to a de Bruijn level $dst-1-x$ and returns a function which, when applied to dst', converts this level back to an index. It is expected that dst' is greater than or equal to *dst*. The idea is to exploit the fact that de Bruijn levels are unaffected by the growth of the name space that they inhabit.

The relocatable term *lambda t*, once instructed of the height *dst* of the target name space, increases *dst* by 1, so as to make room for one more variable, applies the function t to a relocatable representation of the variable 0, and applies the resulting relocatable term to *dst*, so as to obtain a term.

The identity function is represented by the relocatable term *lambda* $(x \Rightarrow x)$, which (as can be checked by unfolding) is a constant function which ignores its argument *dst* and always returns the term $\lambda 0$.

The representation predicate of Definition 3 can be used to give specifications to the functions *var*, *lambda*, and *app*. For instance, provided $x < dst$ holds, the relocatable term *var dst x* represents the term x at time *dst*. We omit the specifications of *lambda* and *app*.

$$\begin{array}{lll} \mathit{interpret}\ (\mathit{src},m,\mathit{dst})\ x = \mathit{dst} - 1 - \mathit{find}\ (\mathit{src}-1-x)\ m & \text{if } x < \mathit{src} \\ \mathit{interpret}\ (\mathit{src},m,\mathit{dst})\ x = x + \mathit{dst} - \mathit{src} & \text{otherwise} \end{array}$$

Fig. 5 Interpretation of a triple (src,m,dst) as a renaming

$$\begin{array}{rcl} \mathit{var} & : & \text{nat} \to \text{nat} \to \text{rterm} \\ \mathit{var}\ \mathit{dst}\ x & = & \mathit{dst}' \Rightarrow \mathit{dst}' - 1 - (\mathit{dst} - 1 - x) \\[4pt] \mathit{lambda} & : & (\text{rterm} \to \text{rterm}) \to \text{rterm} \\ \mathit{lambda}\ t & = & \mathit{dst} \Rightarrow \lambda(\mathit{let}\ \mathit{dst} = \mathit{dst}+1\ \mathit{in}\ t\ (\mathit{var}\ \mathit{dst}\ 0)\ \mathit{dst}) \\[4pt] \mathit{app} & : & \text{rterm} \to \text{rterm} \to \text{rterm} \\ \mathit{app}\ t_1\ t_2 & = & \mathit{dst} \Rightarrow (t_1\ \mathit{dst})\ @\ (t_2\ \mathit{dst}) \end{array}$$

Fig. 6 Construction functions for relocatable terms

$$\begin{array}{rcll} \mathit{rapply} & : & \text{rcont} \to \text{rterm} \to \text{rterm} & \\ \mathit{rapply}\ (\mathsf{o}\ k)\ v & = & \mathit{app}\ k\ v & \text{– an object-level application} \\ \mathit{rapply}\ (\mathsf{m}\ \kappa)\ v & = & \kappa\ v & \text{– a metalevel application} \\[4pt] \mathit{rreify} & : & \text{rcont} \to \text{rterm} & \\ \mathit{rreify}\ k & = & k & \text{– a no-op} \\ \mathit{rreify}\ \kappa & = & \mathit{lambda}\ \kappa & \text{– a two-level } \eta\text{-expansion} \end{array}$$

Fig. 7 Operations on relocatable continuations

$$\begin{array}{rcl} \mathit{svar} & : & \text{nat} \to (\text{env} \to \text{rterm}) \\ \mathit{svar}\ x\ (\mathit{src},m) & = & \mathit{dst} \Rightarrow \mathit{interpret}\ (\mathit{src},m,\mathit{dst})\ x \\[4pt] \mathit{sbind} & : & (\text{env} \to \text{rterm}) \to (\text{env} \to \text{rterm}) \\ \mathit{sbind}\ t\ (\mathit{src},m) & = & \mathit{dst} \Rightarrow \mathit{let}\ (\mathit{src},m,\mathit{dst}) = (\mathit{src}+1, \mathit{add}\ \mathit{src}\ \mathit{dst}\ m, \mathit{dst}+1)\ \mathit{in}\ t\ (\mathit{src},m)\ \mathit{dst} \\[4pt] \mathit{slambda} & : & \text{env} \to (\text{env} \to \text{rterm}) \to \text{rterm} \\ \mathit{slambda}\ \rho\ t & = & \mathit{dst} \Rightarrow \lambda(\mathit{sbind}\ t\ \rho\ \mathit{dst}) \\[4pt] \mathit{slet} & : & \text{env} \to \text{rterm} \to (\text{env} \to \text{rterm}) \to \text{rterm} \\ \mathit{slet}\ \rho\ t_1\ t_2 & = & \mathit{dst} \Rightarrow \mathsf{let}\ (t_1\ \mathit{dst})\ \mathsf{in}\ (\mathit{sbind}\ t_2\ \rho\ \mathit{dst}) \\[4pt] \wr \cdot \mid \cdot \wr & : & \text{term} \to \text{env} \to \text{rterm} \\ \wr x \mid \rho \wr & = & \mathit{svar}\ x\ \rho \\ \wr \lambda t \mid \rho \wr & = & \mathit{slambda}\ \rho\ (\rho \Rightarrow \mathit{lambda}\ (k \Rightarrow \wr t \mid \rho \wr \{ \mathsf{o}\ k \})) \\[4pt] \wr \cdot \mid \cdot \wr \{ \cdot \} & : & \text{term} \to \text{env} \to \text{rcont} \to \text{rterm} \\ \wr v \mid \rho \wr \{ c \} & = & \mathit{rapply}\ c\ \wr v \mid \rho \wr \\ \wr t_1\ @\ t_2 \mid \rho \wr \{ c \} & = & \wr t_1 \mid \rho \wr \{ \mathsf{m}\ v_1 \Rightarrow \wr t_2 \mid \rho \wr \{ \mathsf{m}\ v_2 \Rightarrow \mathit{app}\ (\mathit{app}\ v_1\ v_2)\ (\mathit{rreify}\ c) \} \} \\ \wr \mathsf{let}\ t_1\ \mathsf{in}\ t_2 \mid \rho \wr \{ c \} & = & \wr t_1 \mid \rho \wr \{ \mathsf{m}\ v_1 \Rightarrow \mathit{slet}\ \rho\ v_1\ (\rho \Rightarrow \wr t_2 \mid \rho \wr \{ c \}) \} \end{array}$$

Fig. 8 A higher-order, one-pass, call-by-value CPS transformation — formal & efficient version

7.3 Relocatable continuations

In our initial informal presentation of the CPS transformation (§2), we have used a higher-order style, where terms-with-a-hole are represented as metalevel functions. Then (§3), we have switched to a first-order style, where they are represented as syntax, because this style is easier to reason about. We now switch back to a higher-order style, because representing a term-with-a-hole as a metalevel function is efficient. The operation of filling the hole with a value is just metalevel function application; there is no need for a substitution operation. Provided this operation is ever performed at most once, this approach is profitable. In

the CPS transformation, every term-with-a-hole is filled exactly once, so this condition is satisfied.

A **relocatable continuation** c is either a relocatable term k (of type rterm) or a metalevel function κ (of type rterm $\to$ rterm). We write $c ::= \mathsf{o}\ k \mid \mathsf{m}\ \kappa$. We write "rcont" for the type of relocatable continuations.

The meaning of a relocatable continuation is made explicit by a 3-place predicate whose parameters respectively have types rcont, cont, and nat, as follows.

Definition 4 The assertion that a relocatable continuation c **represents** a continuation c' **at time** dst is defined by the following two rules:

1. If k represents k' at time dst, then $\mathsf{o}\ k$ represents $\mathsf{o}\ k'$ at time dst.
2. If, for all i, for all values v and v' such that v represents v' at time $dst+i$,
   $\kappa\ v$ represents $\kappa'[v' \cdot (+i)]$,
   then $\mathsf{m}\ \kappa$ represents $\mathsf{m}\ \kappa'$ at time dst.

We do not attempt to explain in detail this admittedly slightly cryptic definition. Let us just state the following two lemmas, which confirm that everything works as expected. Lemma 12 states that relocatable continuations are indeed relocatable. (Its statement has the same structure as that of Lemma 11.) Lemma 13 states that the operations *rapply* and *rreify*, defined in Figure 7, are correct implementations of *apply* and *reify*.

Lemma 12 (Relocation) *If c represents c' at time dst, then c represents $\uparrow^i c'$ at time $dst+i$.*

Lemma 13 *If c represents c' at time dst, then:*

1. *If v represents v' at time dst, then rapply c v represents apply c' v' at time dst.*
2. *rreify c represents reify c' at time dst.*

7.4 An efficient implementation of the CPS transformation

An efficient formulation of the CPS transformation is given in Figure 8. The value transformation, written $\wr v \mid \rho \wr$, and the term transformation, written $\wr t \mid \rho \wr \{ c \}$, are defined in a mutually recursive manner. We write ρ for a pair (src, m), and we write env for the type of such a pair. The type of the value transformation function $\wr \cdot \mid \cdot \wr$ is term $\to$ env $\to$ rterm, which means that this function must be successively applied to a source value v, to a pair (src, m), and to a number dst, producing a transformed term. Together, (src, m, dst) represent a renaming (§7.1), that is, a total function of variables in the source name space to variables in the target name space. Yet, things have been arranged so that (src, m) and dst are passed separately, in two distinct phases; the partial application $\wr v \mid \rho \wr$ is a relocatable term.

We do not explain the definition in detail. Let us note that *svar* applies the renaming (src, m, dst) to a source variable, yielding a target variable. *sbind* is used when a binder in the source term gives rise to a binder in the target term; then, the current renaming σ becomes $\Uparrow \sigma$. This is the case in *slambda* and *slet*, which are used when a λ or let binder in the source term gives rise to a λ or let binder in the target term.

The following lemma states that this formulation of the CPS transformation is equivalent to our earlier formulation (§3). Its proof, by induction on the term t, is not difficult.

Lemma 14 (Correctness) *If (src, m, dst) represents σ and if c represents c' at time dst, then the relocatable term $\wr t \mid (src, m) \wr \{ c \}$ represents the term $[\![t[\sigma]]\!] \{ c' \}$ at time dst.*

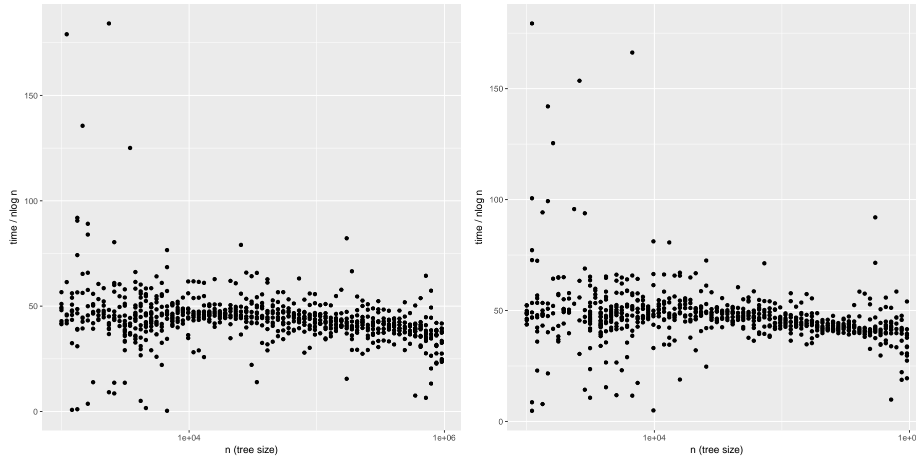

Fig. 9 Running time divided by $n\log n$, as a function of n. The horizontal scale is logarithmic. At left, random terms; at right, "right-leaning" random terms, whose `let` constructs have a left-hand side of bounded size.

As a corollary, we find that the two formulations of the CPS transformation coincide.

Lemma 15 **(Coincidence)** *Let us write ρ_0 for the initial environment $(0, empty)$ and rdone for the empty-context relocatable continuation* m $(t \Rightarrow t)$*. Then, for every term t, the terms* $\wr t \mid \rho_0 \int \{ rdone \}\ 0$ *and* $[\![t]\!] \{ done \}$ *coincide.*

This formulation of the CPS transformation, whose complete code appears in Figures 5 to 8, is efficient. We argue informally that its worst-case time complexity is $O(n\log n)$, where n is the size of the source term. Indeed, no lifting or substitution operations whatsoever are involved. The dictionary operations *find* and *add* have worst-case time complexity $O(\log n)$. Every other elementary operation, including allocations of abstract syntax tree nodes and closure allocations, has worst-case time complexity $O(1)$. Finally, every closure is invoked exactly once, so the cost of executing this closure can be charged to the site where this closure is allocated. It should then be evident that the worst-case time complexity of the transformation is $O(n\log n)$. This claim would arguably deserve a formal proof, which we leave to future work!

We have extracted OCaml code from this definition and have experimentally confirmed that the time complexity of the transformation, applied to randomly generated closed terms, seems to be $\Theta(n\log n)$. Indeed, the running time (measured by Jane Street's `core_bench` library, which attempts to even out the effect of GC), divided by $n\log n$, seems to give rise to a constant function of n (Figure 9).

8 Related work

The CPS transformation and its formulations Reynolds (1993) recounts the discoveries of continuations and of the CPS transformation. The earliest widely-cited accounts of the CPS transformation, in a two-pass formulation, are Fischer's (1972, 1993) and Plotkin's (1975). Danvy and Filinski (1992) offer a one-pass, higher-order formulation of the transformation. Sabry and Felleisen (1993) propose a "compactifying" transformation, whose formulation is one-pass and first-order. It is not compositional, that is, not defined directly by induction on

the structure of terms. Indeed, before a term can be transformed, it must be identified as one of (1) a value or (2) a neutral term in an evaluation context or (3) a β-redex in an evaluation context. Danvy and Nielsen (2004) explain that the extra "compactifying" power of Sabry and Felleisen's transformation can be obtained by turning n-ary applications of curried n-ary functions into cascades of let constructs prior to a conventional CPS transformation. Lawall and Danvy (1993) show that the CPS transformation can be understood as the composition of three steps, namely naming intermediate results, introducing continuations, and deciding the evaluation order. Finally, Danvy and Nielsen (2003) present a one-pass, compositional, first-order formulation of the transformation. A shortcoming of this formulation is that it is redundant: the transformation of applications requires distinguishing four cases, and the transformation of n-tuples would require distinguishing 2^n cases. Another shortcoming is that, as illustrated earlier (§6), extending it with support for let constructs results in the production of administrative redexes. In contrast, the formulation proposed in this paper (§3) is also one-pass, compositional, and first-order, but is just as elegant as Danvy and Filinski's higher-order formulation. In fact, it is arguably just a new reading of Danvy and Filinski's formulation with first-order glasses.

Type-preserving CPS transformations In this paper, we have carried out a purely untyped study of the CPS transformation. The type-preserving character of the transformation has been studied by numerous researchers, beginning with Meyer and Wand (1985) for the simply-typed λ-calculus and including Harper and Lillibridge (1993), Morrisett *et al.* (1999), Barthe *et al.* (1999), and Shao *et al.* (2005), for ever-richer typed calculi. Subsequently, many authors have shown how the property of type preservation can be statically verified by the type-checker of the metalanguage in which the transformation is defined (Chen and Xi, 2003, Linger and Sheard, 2004, Chlipala, 2007, 2008, Guillemette and Monnier, 2007, 2008, Savary Belanger et al, 2015).

Mechanized accounts of the CPS transformation The POPLmark challenge (Aydemir et al, 2005) has drawn attention to the fact that formalizing the metatheory of programming languages is now feasible and desirable, but sometimes remains challenging, due in large part to the difficulty of dealing with binding structure in a pleasant manner. The present work can be viewed as another case study in the spirit of the POPLmark challenge. In our experience, thanks to Autosubst (Schäfer et al, 2015), it is now relatively easy to reason in Coq about terms in de Bruijn's representation. We believe that the statements of the auxiliary lemmas in §4 have been simplified as a result of Autosubst's view of substitutions as total functions. The proofs of these lemmas would have been much less tractable without Autosubst.

Minamide and Okuma (2003) formalize three formulations of the CPS transformation, namely Plotkin's (1975), Danvy and Nielsen's (2003), and Danvy and Filinski's (1992), in the proof assistant Isabelle/HOL. Because they find de Bruijn indices "difficult to manage", they prefer to use traditional named variables. This causes them to run into a number of difficulties, as they must explicitly keep track of the uniqueness or freshness of variables and explicitly account for α-equivalence. They also report problems in dealing with the fact that the latter two formulations are functions of two arguments: when applied to a term, they yield a function of a continuation to a term. At the time, this "defeated Isabelle's automated tactics." To work around this, Minamide and Okuma reformulate these transformations as functions which, when applied to a term, produce a context, that is, a term with a single unnamed hole. This differs from our view of a metalevel continuation as a term with one named hole, which arguably is more natural. As reported earlier (§6), Minamide and Okuma

extend Danvy and Nielsen's transformation with support for let constructs. They report further problems with the uniqueness of bound variables, but are nevertheless able to establish that the simple simulation diagram holds.

Tian (2006) uses Twelf to formalize a higher-order, CPS transformation. His source and target calculi differ: whereas the source calculus is a variant of the λ-calculus (with an explicit redundancy between values and expressions), the target calculus is a nonstandard continuation-passing style calculus. Although the construct $\mathsf{App}(V_1, V_2, k)$, where k denotes a metalevel abstraction $x \Rightarrow E$, can informally be read as $\mathsf{let}\ x = V_1 \ @\ V_2 \ \mathsf{in}\ E$, the construct $\mathsf{Lam}(x,k).E$, where k also denotes a metalevel abstraction, is very much nonstandard: it is a binder at a higher type. Accordingly, the operational semantics of the target calculus involves a substitution at a higher type (rule *red_app*, Figure 2). This calculus apparently cannot be faithfully modeled using ordinary first-order terms and substitutions (of terms for variables). We believe that it could be modeled by using hereditary substitutions (of contexts for context variables).

Dargaye and Leroy (2007) verify a one-pass CPS transformation, with support for let constructs (among other features), which we believe is equivalent to Danvy and Filinski's properly-tail-recursive transformation. They propose a first-order, one-pass formulation, but, departing from Danvy and Nielsen's approach, they use a "smart application" constructor, written $@_\beta$, which reduces administrative redexes on the fly. This seems to be another way of avoiding the shortcomings of Danvy and Nielsen's formulation. However, the fact that smart application does not commute with substitution creates a slight complication in the proof. Instead of directly establishing the correctness of the transformation, Dargaye and Leroy first establish the correctness of a naïve transformation, which does not eliminate administrative redexes, then prove that the two transformations are the same, up to parallel reductions (Dargaye and Leroy, 2007, Lemma 6). Thus, although they follow a different technical path, they end up exploiting parallel reduction, probably for the same fundamental reason as we do. They use a big-step operational semantics, so do not establish a simulation diagram. Nevertheless, their correctness statement (Dargaye and Leroy, 2007, Theorem 2) is closely related to ours (Lemma 8, item 1).[3] Dargaye and Leroy choose to "avoid some of the difficulties associated with standard de Bruijn indices" by using two distinct namespaces for the source variables and for the variables introduced by the translation. This approach seems effective, but makes the statements of the substitution lemmas heavier (Dargaye and Leroy, 2007, Lemmas 1 and 2) and requires performing a (verified) conversion from the two-namespace calculus back to a standard λ-calculus. Perhaps Autosubst could make Dargaye and Leroy's approach more lightweight; anyway, we show that a direct transformation is tractable.

Chlipala (2007) presents a verified type-preserving compiler from the simply-typed λ-calculus to assembly language, which includes a transformation into CPS form. He uses dependent types to ensure that object-level terms are well-scoped and well-typed. Variables are represented as de Bruijn indices. The proof of semantic preservation is carried out by defining (type-directed) interpretations of both the source and target languages into the metalanguage and by defining a (type-directed) logical relation that relates these interpretations.

[3] It might seem as though the parallel reduction sequence $v' \Longrightarrow^\star_{\text{cbv}} (\!|v|\!)$ in our Lemma 8 is stated in the reverse direction, compared with Dargaye and Leroy's Theorem 2. The confusion is cleared by noting that the function Ψ in their Theorem 2 denotes the *nonoptimizing* transformation of values. Thus, our statement and their statement are indeed slightly different, and, by combining them, one finds that the value v' lies somewhere on a parallel reduction path from the nonoptimized translation of v to the optimized translation of v. Dargaye and Leroy inadvertently use the metavariable Ψ to refer to both the nonoptimizing and optimizing versions of the value transformation.

This is quite different from our approach, which is based on an operational semantics and does not exploit type structure. In a subsequent paper (2008), Chlipala moves away from de Bruijn indices and instead proposes “parametric higher-order abstract syntax”, a form of higher-order abstract syntax (Pfenning and Elliott, 1988) where parametric polymorphism is used to ensure that variables are considered opaque and thereby eliminate “exotic terms”. A type-preserving CPS transformation for System F is presented. Semantics preservation is proved via a “foundational type-theoretic semantics”, as in Chlipala’s previous paper (2007).

9 Conclusion

In this paper, we have proposed a new first-order, one-pass, compositional formulation of the call-by-value CPS transformation. This new formulation can be viewed as a new reading of Danvy and Filinski’s formulation (1992). We have also proposed a new higher-order, one-pass, compositional formulation, which leads to an efficient implementation. We have proved that these formulations define the same transformation and that this transformation is semantics-preserving. We have clarified which simulation diagram must be used in the latter proof, as Danvy and Filinski’s simple simulation diagram (1992, Lemma 3) does not hold in the presence of a let construct. We have found that, thanks to Autosubst (Schäfer et al, 2015), formal reasoning about terms in de Bruijn’s representation is tractable. Our definitions and proofs have been machine-checked using Coq and are available online (Pottier, 2017). We hope that some of the techniques illustrated in this paper can be useful in teaching and in the construction of verified compilers, beyond the CPS transformation.

We found the efficient formulation proposed in §7 rather difficult to discover. Somewhat distressingly, although its proof of correctness is reasonably simple, we still find it difficult to explain why and how this code works. A related technical remark is that we were not able to assign fully satisfactory specifications to the auxiliary functions *lambda* (Figure 6) and *sbind*, *slambda*, and *slet* (Figure 8). Instead, in the proof of Lemma 14, we expand away these auxiliary functions, and reason directly about the resulting code. As another remark, because this code involves higher-order functions, it is hard to understand its operational behavior, that is, to understand exactly which instructions are carried out and in which order. Perhaps studying a defunctionalized version of this code would shed new light on it.

Future work might include scaling up our definitions and proofs so as to handle a richer source language, including products, sums, and so on. One might also wish to formalize the call-by-name CPS transformation. At a metatheoretic level, it would be worth investigating whether and how metatheoretic definitions and statements can be systematically translated from one representation of names to another: say, from the nominal representation to the de Bruijn’s representation, and vice-versa. Can one systematically translate freshness side conditions into end-of-scope operators, and vice-versa? Can one somehow compensate for the fact that the nominal representation relies on unordered contexts, whereas de Bruijn’s representation relies on ordered contexts?

References

Aydemir BE, Bohannon A, Fairbairn M, Foster JN, Pierce BC, Sewell P, Vytiniotis D, Washburn G, Weirich S, Zdancewic S (2005) Mechanized metatheory for the masses: The POPLMARK challenge. In: Theorem Proving in Higher Order Logics (TPHOLs), Springer, Lecture Notes in Computer Science, vol 3603, pp 50–65

Barthe G, Hatcliff J, Sørensen MH (1999) CPS translations and applications: The cube and beyond. Higher-Order and Symbolic Computation 12(2):125–170

Barthe G, Forest J, Pichardie D, Rusu V (2006) Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In: Functional and Logic Programming, Springer, Lecture Notes in Computer Science, vol 3945, pp 114–129

Bird R, Paterson R (1999) de Bruijn notation as a nested datatype. Journal of Functional Programming 9(1):77–91

de Bruijn NG (1972) Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. Indag Math 34(5):381–392

Chen C, Xi H (2003) Meta-programming through typeful code representation. In: International Conference on Functional Programming (ICFP), pp 275–286

Chlipala A (2007) A certified type-preserving compiler from lambda calculus to assembly language. In: Programming Language Design and Implementation (PLDI), pp 54–65

Chlipala A (2008) Parametric higher-order abstract syntax for mechanized semantics. In: International Conference on Functional Programming (ICFP), pp 143–156

Chlipala A (2013) Certified Programming and Dependent Types. MIT Press

Crary K (2009) A simple proof of call-by-value standardization. Technical Report CMU-CS-09-137, Carnegie Mellon University

Danvy O, Filinski A (1992) Representing control: A study of the CPS transformation. Mathematical Structures in Computer Science 2(4):361–391

Danvy O, Nielsen LR (2003) A first-order one-pass CPS transformation. Theoretical Computer Science 308(1–3):239–257

Danvy O, Nielsen LR (2004) CPS transformation of beta-redexes. Technical Report RS-04-39, BRICS

Dargaye Z, Leroy X (2007) Mechanized verification of CPS transformations. In: Logic for Programming Artificial Intelligence and Reasoning (LPAR), Springer, Lecture Notes in Artificial Intelligence, vol 4790, pp 211–225

Fischer MJ (1972) Lambda calculus schemata. In: Proceedings of the ACM Conference on Proving Assertions About Programs, pp 104–109

Fischer MJ (1993) Lambda-calculus schemata. Lisp and Symbolic Computation 6(3–4):259–288

Guillemette LJ, Monnier S (2007) Type-safe code transformations in Haskell. Electronic Notes in Theoretical Computer Science 174(7):23–39

Guillemette LJ, Monnier S (2008) A type-preserving compiler in Haskell. In: International Conference on Functional Programming (ICFP), pp 75–86

Harper R, Lillibridge M (1993) Polymorphic type assignment and CPS conversion. Lisp and Symbolic Computation 6(3–4):361–380

Hendriks D, van Oostrom V (2003) Adbmal. In: International Conference on Automated Deduction (CADE), Springer, Lecture Notes in Computer Science, vol 2741, pp 136–150

Huffman B, Urban C (2010) A new foundation for Nominal Isabelle. In: Interactive Theorem Proving (ITP), Springer, Lecture Notes in Computer Science, vol 6172, pp 35–50

Lawall JL, Danvy O (1993) Separating stages in the continuation-passing style transformation. In: Principles of Programming Languages (POPL), pp 124–136

Linger N, Sheard T (2004) Programming with static invariants in Ωmega, unpublished

Meyer AR, Wand M (1985) Continuation semantics in typed lambda-calculi. In: Logics of Programs, Springer, Lecture Notes in Computer Science, vol 193, pp 219–224

Minamide Y, Okuma K (2003) Verifying CPS transformations in Isabelle/HOL. In: ACM Workshop on Mechanized Reasoning about Languages with Variable Binding

Morrisett G, Walker D, Crary K, Glew N (1999) From system F to typed assembly language. ACM Transactions on Programming Languages and Systems 21(3):528–569

Okasaki C (1999) Purely Functional Data Structures. Cambridge University Press

Pfenning F, Elliott C (1988) Higher-order abstract syntax. In: Programming Language Design and Implementation (PLDI), pp 199–208

Pitts AM (2005) Alpha-structural recursion and induction. In: Theorem Proving in Higher Order Logics (TPHOLs), Springer, Lecture Notes in Computer Science

Plotkin GD (1975) Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science 1(2):125–159

Pottier F (2017) Proofs about the cps transformation. `https://gitlab.inria.fr/fpottier/cps`

Reynolds JC (1993) The discoveries of continuations. Lisp and Symbolic Computation 6(3–4):233–248

Sabry A, Felleisen M (1993) Reasoning about programs in continuation-passing style. Lisp and Symbolic Computation 6(3–4):289–360

Savary Belanger O, Monnier S, Pientka B (2015) Programming type-safe transformations using higher-order abstract syntax. Journal of Formalized Reasoning 8(1)

Schäfer S, Tebbi T, Smolka G (2015) Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In: Interactive Theorem Proving (ITP), Springer, Lecture Notes in Computer Science, vol 9236, pp 359–374

Shao Z, Trifonov V, Saha B, Papaspyrou N (2005) A type system for certified binaries. ACM Transactions on Programming Languages and Systems 27(1):1–45

Sozeau M (2007) Program-ing finger trees in Coq. In: International Conference on Functional Programming (ICFP), pp 13–24

Takahashi M (1995) Parallel reductions in λ-calculus. Information and Computation 118(1):120–127

Tian YH (2006) Mechanically verifying correctness of CPS compilation. In: Computing: The Australasian Theory Symposium (CATS), Australian Computer Society, CRPIT, vol 51, pp 41–51

Urban C (2008) Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning 40(4):327–356