Mechanized verification of CPS transformations

Zaynah Dargaye, Xavier Leroy

Mechanized Verification of CPS Transformations

Zaynah Dargaye and Xavier Leroy

INRIA Paris-Rocquencourt
B.P. 105, 78153 Le Chesnay, France
Zaynah.Dargaye@inria.fr, Xavier.Leroy@inria.fr

Abstract. Transformation to continuation-passing style (CPS) is often performed by optimizing compilers for functional programming languages. As part of the development and proof of correctness of a compiler for the mini-ML functional language, we have mechanically verified the correctness of two CPS transformations for a call-by-value λ-calculus with n-ary functions, recursive functions, data types and pattern-matching. The transformations generalize Plotkin's original call-by-value transformation and Danvy and Nielsen's optimized transformation, respectively. We used the Coq proof assistant to formalize the transformations and conduct and check the proofs. Originalities of this work include the use of big-step operational semantics to avoid difficulties with administrative redexes, and of two-sorted de Bruijn indices to avoid difficulties with α-conversion.

1 Introduction

Continuation-passing style (CPS) is a programming style in the λ-calculus and related functional languages where a function never returns directly the result of its computations, but instead passes it to another function, the *continuation*, received as an extra argument and representing the meaning of the rest of the program. For instance, the successor function, written $\lambda x.\ x + 1$ in direct style, becomes $\lambda x.\lambda k.\ k(x+1)$ in continuation-passing style, where k is the continuation parameter. Programs can be systematically translated to semantically equivalent programs in CPS using a variety of CPS transformation algorithms (see Sect. 2 for examples).

CPS and the related CPS transformations play an important role in three domains relevant to programming languages: semantics, programming, and compilation.

As a semantic device, CPS makes it possible to use the pure λ-calculus (without a fixed evaluation strategy) as a meta-language to describe faithfully the semantics of functional or imperative programming languages. After translation to CPS, the evaluation strategy of these languages is encoded in the structure of the resulting λ-term. Additionally, CPS makes it easy to give formal semantics to advanced control structures such as exceptions, backtracking, coroutines and control operators.

As a programming device, CPS enables functional programmers to define advanced, application-specific control structures such as coroutines or non-blind

backtracking. These control structures need not be supported natively by the programming language.

As a compilation device, programs in CPS lend themselves to aggressive optimizations that are significantly harder to perform on direct-style programs. CPS has several features that facilitate optimizations: all intermediate results are named, and compile-time β-reductions are always semantically valid. Several optimizing compilers for functional languages, such as Orbit Scheme [15], Standard ML of New Jersey [2], and SML.NET [13], use CPS as an intermediate language.

In this paper, we describe the formal verification, using the Coq proof assistant [7, 4], of the correctness (semantic preservation) of two CPS transformations for a realistic, pure, call-by-value functional language. This language features n-ary functions, recursive functions, and ML/Haskell-style data types and pattern-matching. The two CPS transformations are extensions of Plotkin's original call-by-value transformation [22] and Danvy and Nielsen's optimized transformation [8, 9], respectively.

This work is part of a larger project that aims at mechanically verifying the correctness of a whole compiler for mini-ML, a pure, call-by-value functional language rich enough to be used as a target language for automatic extraction of functional programs from Coq specifications [19]. In a context where formal methods are increasingly being applied to critical software, it becomes important to guarantee that compilers preserve the semantics of the programs they compile: a bug in a compiler could result in incorrect executable code being produced from correct, formally verified source programs. One way to obtain this guarantee is to formally verify the compiler itself, using theorem provers to prove that it is correct, i.e. preserves the semantics of source programs. Several non-trivial compilers have been formally verified along these lines, for assembly languages [21], imperative languages [16, 18, 5], object-oriented languages [14] and functional languages [6]. The work presented here is part of the development and verification of a front-end compiler from mini-ML to the Cminor intermediate language [18]. Our long-term plan is to combine this front-end with the verified back-end for Cminor described in [18] and with a future verification of the extraction mechanism from Coq functional specifications to mini-ML to obtain a trusted execution path for programs directly written in the Coq specification language.

Related Work

Many on-paper proofs of correctness for various CPS transformations have been published already, starting with Plotkin's seminal article [22]. We are aware of three earlier on-machine formalizations and correctness proofs for CPS transformations: one by Minamide and Okuma [20], using Isabelle/HOL; one by Tian [24], using Twelf; and one by Chlipala [6], using Coq.

A recurring difficulty in mechanizing programming language semantics, type systems and program transformations is the handling of binders and α-conversion

(the fact that $\lambda x.x$ and $\lambda y.y$ are equivalent terms). Most existing proof assistants provide no native support for working with terms modulo α-conversion like we routinely do on paper. (The only exception is Urban's Isabelle/HOL implementation of nominal logic [25].) The POPLmark challenge [3] gives an excellent summary of the difficulties this raises when mechanizing properties of programming languages and of the known techniques to circumvent these difficulties: de Bruijn indices, higher-order abstract syntax, locally nameless representations, ... In the case of CPS transformations, Minamide and Okuma use named variables with no α-conversion for Plotkin's naive CPS transformation, and with explicit renamings for Danvy and Nielsen's optimized transformation. Tian uses higher-order abstract syntax to reason about Danvy and Nielsen's CPS transformation. Like Chlipala, we use de Bruijn indices [11] to provide unique representatives for λ-terms. We avoid some of the difficulties associated with standard de Bruijn indices by using two kinds of de Bruijn indices, independently numbered: one for source variables and one for continuation variables introduced by the CPS transformation.

Earlier work also differs on the kind of operational semantics used to prove the correctness of CPS transformations. Following Plotkin's original proof, Minamide and Okuma use small-step semantics, while Tian uses a combination of big-step semantics and small-step semantics for the source and target languages, respectively, and Chlipala uses a form of denotational semantics directed by the types of the simply-typed λ-calculus. We use (untyped) big-step semantics for the source and target languages. A strength of big-step semantics is that it avoids the well-known difficulties caused by administrative redexes in Plotkin's original, small-step proof of CPS transformations. A weakness of big-step semantics is that it captures only terminating executions, and therefore cannot be used to prove semantic preservation for diverging source programs. This limitation is unproblematic in our intended usage scenario, since programs extracted from Coq functional specifications are strongly normalizing.

Finally, earlier mechanizations handle only the pure λ-calculus, while we cover a larger, more realistic functional language including n-ary functions, recursive functions, data types and pattern-matching. These extensions are conceptually easy but technically not entirely obvious. Mechanizing the correctness proof is especially useful to ensure that we do not overlook the small difficulties raised by these extensions.

Outline

The remainder of this paper is organized as follows. Section 2 reviews some of the known CPS transformations. Section 3 defines the source and target languages for our transformations. We define and outline the correctness proof of two CPS transformations in Sect. 4 and 5. Section 6 gives some practical information on the Coq mechanization of these results. Concluding remarks are given in Sect. 7.

2 Examples of CPS Transformations

We start by reviewing some of the many known variants of CPS transformation for call-by-value λ-calculus. One of the earliest and simplest transformations is that of Plotkin [22]:

$$\begin{aligned}
[\![x]\!]_1 &= \lambda k.\ k\ x \\
[\![\lambda x.M]\!]_1 &= \lambda k.\ k\ (\lambda x.\ [\![M]\!]_1) \\
[\![M\ N]\!]_1 &= \lambda k.\ [\![M]\!]_1\ (\lambda m.\ [\![N]\!]_1\ (\lambda n.\ m\ n\ k))
\end{aligned}$$

Each source term is transformed into an abstraction $\lambda k \ldots$ over the continuation for this term. A weakness of this transformation is that it generates many *administrative redexes*, that is, β-redexes that correspond to no redex in the original source term. For instance, the translation of $x\ y$ contains four such redexes, outlined below:

$$\begin{aligned}
[\![x\ y]\!]_1 &= \lambda k.\ \underline{(\lambda k.\ k\ x)\ (\lambda m.\ (\lambda k.\ k\ y)\ (\lambda n.\ m\ n\ k))} \\
&\xrightarrow{\beta} \lambda k.\ \underline{(\lambda m.\ (\lambda k.\ k\ y)\ (\lambda n.\ m\ n\ k))\ x} \\
&\xrightarrow{\beta} \lambda k.\ \underline{(\lambda k.\ k\ y)\ (\lambda n.\ x\ n\ k)} \xrightarrow{\beta} \lambda k.\ \underline{(\lambda n.\ x\ n\ k)\ y} \xrightarrow{\beta} \lambda k.\ x\ y\ k
\end{aligned}$$

When CPS transformation is used as part of a compiler, these administrative redexes introduce inefficiencies that must be eliminated by a later pass of compile-time β-reduction.

The following variant of Plotkin's transformation avoids the generation of some, but not all administrative redexes. Here, instead of λ-abstracting over the continuation variable k, we turn k into an additional parameter of the (mathematical) function that defines the translation. The translation therefore becomes $[\![M]\!]_2 \rhd k$ where M is the source term and k a continuation term.

$$\begin{aligned}
[\![x]\!]_2 \rhd k &= k\ x \\
[\![\lambda x.M]\!]_2 \rhd k &= k\ (\lambda x.\lambda k.\ [\![M]\!]_2 \rhd k) \\
[\![M\ N]\!]_2 \rhd k &= [\![M]\!]_2 \rhd \lambda m.\ [\![N]\!]_2 \rhd \lambda n.\ m\ n\ k
\end{aligned}$$

We now have $[\![x\ y]\!]_2 \rhd k = (\lambda m.\ (\lambda n.\ m\ n\ k)\ y)\ x$, which contains only two administrative β-redexes.

Danvy and Nielsen [8] present the following refinement of the two-place translation above that avoids generating any administrative redex. It distinguishes λ-terms that are *atoms* $A, B ::= x \mid \lambda x.M$ from the other λ-terms $P, Q ::= M\ N$. The transformation is presented as two mutually recursive functions, $\Psi_3(A)$ for atoms A and $[\![M]\!]_3 \rhd k$ for arbitrary terms M.

$$\begin{aligned}
[\![A]\!]_3 \rhd k &= k\ \Psi_3(A) & \Psi_3(x) &= x \\
[\![A\ B]\!]_3 \rhd k &= \Psi_3(A)\ \Psi_3(B)\ k & \Psi_3(\lambda x.M) &= \lambda x.\lambda k.\ [\![M]\!]_3 \rhd k \\
[\![P\ B]\!]_3 \rhd k &= [\![P]\!]_3 \rhd \lambda p.\ p\ \Psi_3(B)\ k \\
[\![A\ Q]\!]_3 \rhd k &= [\![Q]\!]_3 \rhd \lambda q.\ \Psi_3(A)\ q\ k \\
[\![P\ Q]\!]_3 \rhd k &= [\![P]\!]_3 \rhd \lambda p.\ [\![Q]\!]_3 \rhd \lambda q.\ p\ q\ k
\end{aligned}$$

We now have $[\![x\ y]\!]_3 \triangleright k = x\ y\ k$, as desired. However, the case for applications was split in 4 different cases, depending on whether the function and its argument are atoms or not. This combinatorial explosion makes it difficult to extend this transformation to n-ary function applications and data constructor applications.

To circumvent this difficulty, we use (in Sect. 5) the following alternate presentation of Danvy and Nielsen's transformation. We define the "smart application" constructor $@_\beta$ that reduces (on the fly) administrative redexes that would arise if the first argument is a lambda-abstraction and the second argument is an atom:

$$(\lambda x.M)\ @_\beta\ A = M\{x \leftarrow A\} \qquad\qquad M\ @_\beta\ N = M\ N \text{ otherwise}$$

We then use $@_\beta$ instead of regular applications in the variable and abstraction cases of the translation $[\![M]\!]_2 \triangleright k$, obtaining:

$$\begin{aligned}
[\![x]\!]_4 \triangleright k &= k\ @_\beta\ x \\
[\![\lambda x.M]\!]_4 \triangleright k &= k\ @_\beta\ (\lambda x.\lambda k.\ [\![M]\!]_4 \triangleright k) \\
[\![M\ N]\!]_4 \triangleright k &= [\![M]\!]_4 \triangleright \lambda m.\ [\![N]\!]_4 \triangleright \lambda n.\ m\ n\ k
\end{aligned}$$

We have $[\![x\ y]\!]_4 \triangleright k = (\lambda m.\ (\lambda n.\ m\ n\ k)\ @_\beta\ y)\ @_\beta\ x = x\ y\ k$ as expected. More generally, this transformation is extensionally equivalent to that of Danvy and Nielsen: $[\![M]\!]_4 \triangleright k = [\![M]\!]_3 \triangleright k$ if k is a λ-abstraction. Therefore, just like Danvy and Nielsen's transformation, it produces CPS terms that are free of administrative redexes.

3 Source and Target Languages

The source language for the CPS transformation has the following grammar:[1]

Source terms:

M, N, P	$::= x_0 \mid x_1 \mid \ldots$	variables (de Bruijn)
	$\mid \lambda^n.\ M$	function of $n+1$ arguments
	$\mid \mu^n.\ M$	recursive function ($n+1$ args.)
	$\mid M(N_1, \ldots, N_k)$	function application
	$\mid$ `let` M `in` N	bind x_0 to M in N
	$\mid C(N_1, \ldots, N_k)$	data constructor application
	$\mid$ `match` M `with` $\pi_1, \ldots, \pi_k$	pattern-matching

Match cases:

π	$::= C^n \to M$	n is the arity of constructor C

Variables x_i are identified by their de Bruijn indices i. Indices start at 0. The abstraction $\lambda^n.\ M$ has arity $n+1$; it binds variables $x_n, \ldots, x_0$ in M. A recursive abstraction $\mu^n.\ M$ is similar, but in addition x_{n+1} is bound within M to the abstraction itself. In the right-hand side M of a match case $C^n \to M$, variables $x_{n-1}, \ldots, x_0$ are bound to the n arguments of the matched constructor C.

[1] Our Coq development also supports numeric constants and arithmetic and relational operators over numbers. These are omitted in this paper for brevity.

$$\lambda^n.\ M \Rightarrow \lambda^n.\ M \qquad\qquad \mu^n.\ M \Rightarrow \mu^n.\ M$$

$$\frac{M \Rightarrow \lambda^n.\ P \quad N_i \Rightarrow v_i \quad P\{v_n, \ldots, v_0\} \Rightarrow v}{M(N_0, \ldots, N_n) \Rightarrow v}$$

$$\frac{M \Rightarrow \mu^n.\ P \quad N_i \Rightarrow v_i \quad P\{v_n, \ldots, v_0, \mu^n.\ P\} \Rightarrow v}{M(N_0, \ldots, N_n) \Rightarrow v} \qquad \frac{M \Rightarrow v_1 \quad N\{v_1\} \Rightarrow v}{(\texttt{let}\ M\ \texttt{in}\ N) \Rightarrow v}$$

$$\frac{M \Rightarrow C(v_1, \ldots, v_n) \quad \pi_i = (C^n \to N) \quad N\{v_n, \ldots, v_1\} \Rightarrow v}{(\texttt{match}\ M\ \texttt{with}\ \pi_1, \ldots, \pi_k) \Rightarrow v}$$

Fig. 1. Big-step semantics for the source language

The dynamic semantics of this language is given in big-step operational style by the rules in Fig. 1. The rules define the predicate $M \Rightarrow v$, "the term M evaluates to the value v". Values are

$$v ::= \lambda^n.\ M \mid \mu^n.\ M \mid C(v_1, \ldots, v_n).$$

We write $M\{N_0, \ldots, N_k\}$ for the simultaneous substitution of terms $N_0, \ldots, N_k$ for variables $x_0, \ldots, x_k$ in term M. Note the two rules for function application $M(N_0, \ldots, N_n)$, depending on whether M evaluates to a recursive or non-recursive abstraction. In the evaluation rule for the `match` construct, the selected case π_i is the first case that matches constructor C with arity n.

The target language for the CPS transformation is similar, except that it has two kinds of variables, independently numbered by de Bruijn indices: variables x_n correspond to variables already present in the source term, while variables κ_n correspond to variables introduced by the transformation to hold continuations and intermediate evaluation results. The grammar of the target language is therefore:

Target terms:

$M', N', P' ::= x_n$		source-level variables
$\mid \kappa_n$		continuation variables
$\mid \lambda^n.\ M'$		function of $n+1$ arguments
$\mid \mu^n.\ M'$		recursive function ($n+1$ args.)
$\mid M'(N'_1, \ldots, N'_k)$		function application
$\mid \texttt{let}\ M'\ \texttt{in}\ N'$		bind x_0 to M in N
$\mid C(N'_1, \ldots, N'_k)$		data constructor application
$\mid \texttt{match}\ M'\ \texttt{with}\ \pi'_1, \ldots, \pi'_k$		pattern-matching

Match cases:

π'	$::= C^n \to M'$	n is the arity of constructor C

Conventionally, every function takes its continuation as first argument. Therefore, in $\lambda^n.M'$, the first argument is bound to κ_0 in M', and the remaining n arguments are bound to $x_{n-1}, \ldots, x_0$. For a recursive abstraction $\mu^n.M'$, the

$$\lambda^n.\ M' \Rightarrow \lambda^n.\ M' \qquad\qquad \mu^n.\ M' \Rightarrow \mu^n.\ M'$$

$$\frac{M' \Rightarrow \lambda^n.\ P' \qquad N'_i \Rightarrow v_i \qquad P'\{v_0\}\{v_n, \ldots, v_1\} \Rightarrow v}{M'(N'_0, \ldots, N'_n) \Rightarrow v}$$

$$\frac{M' \Rightarrow \mu^n.\ P \qquad N'_i \Rightarrow v_i \qquad P'\{v_0\}\{v_n, \ldots, v_1, \mu^n.\ P'\} \Rightarrow v}{M'(N'_0, \ldots, N'_n) \Rightarrow v}$$

$$\frac{M' \Rightarrow v_1 \qquad N'\{\,\}\{v_1\} \Rightarrow v}{(\texttt{let}\ M'\ \texttt{in}\ N') \Rightarrow v}$$

$$\frac{M' \Rightarrow C(v_1, \ldots, v_n) \qquad \pi'_i = (C^n \to N') \qquad N'\{\,\}\{v_n, \ldots, v_1\} \Rightarrow v}{(\texttt{match}\ M'\ \texttt{with}\ \pi'_1, \ldots, \pi'_k) \Rightarrow v}$$

Fig. 2. Big-step semantics for the target language

variable x_n is additionally bound to the abstraction itself. Match cases and the `let` binding bind source-level variables x_n exactly as in the source language.

The reason why we use two kinds of de Bruijn indices is to simplify the definition of CPS transformations. As observed by Minamide and Okuma [20], if regular de Bruijn indices are used, the transformations need to shift indices of source-level variables to reflect the additional bindings that it inserts. For instance, the naive CPS transformation of $x_i\ x_j$ in regular de Bruijn notation is

$$\lambda^0.\ (\lambda^0.\ x_0(x_{i+2}))\ (\lambda^0.\ (\lambda^0.\ x_0(x_{j+3}))\ (\lambda^0.\ x_1(x_2, x_0)))$$

where the indices i and j of the two source variables are shifted by 2 and 3, respectively. This shifting makes it delicate to define and reason about CPS transformations. Using two kinds of variables avoids this difficulty: the CPS transformation of $x_i\ x_j$ is, then,

$$\lambda^0.\ (\lambda^0.\ \kappa_0(x_i))\ (\lambda^0.\ (\lambda^0.\ \kappa_0(x_j))\ (\lambda^0.\ \kappa_1(\kappa_2, \kappa_0)))$$

The source variables x_i and x_j need not be shifted because all bindings introduced by the translation bind continuation variables $\kappa_0, \kappa_1, \ldots$ but not source variables.

Figure 2 defines the big-step semantics for the target language. The evaluation rules are direct adaptations of those for the source language. We write $M\{N_0, \ldots, N_n\}\{P_0, \ldots, P_p\}$ for the double simultaneous substitution of terms $N_0, \ldots, N_n$ for variables $\kappa_0, \ldots, \kappa_n$ and of terms $P_0, \ldots, P_p$ for variables $x_0, \ldots, x_p$ in term M.

As the semantics use substitution, we will need some standard properties over substitution and the lifting operation such as commutation between lifting and substitution, or neutrality of substitution over closed terms.

The $\Uparrow$ operator denotes lifting of free de Bruijn indices: $\Uparrow_x^n M'$ replaces all x_i variables free in M' by x_{i+n}, and similarly $\Uparrow_\kappa^n M'$ replaces all κ_i variables free in M' by κ_{i+n}.

The following two lemmas about compositions of substitutions play a crucial role in proving semantic preservation for the CPS transformation.

Lemma 1. $(M\{\vec{N}\}\{\vec{P}\})\{\vec{Q}\}\{\vec{R}\} \;=\; (M\{\Uparrow_\kappa^{|\vec{N}|}\Uparrow_x^{|\vec{P}|}\,\vec{Q}\}\{\Uparrow_\kappa^{|\vec{N}|}\Uparrow_x^{|\vec{P}|}\,\vec{R}\})\{\vec{N}\}\{\vec{P}\}$

Lemma 2. $(M\{\Uparrow_\kappa^{|\vec{N}|}\Uparrow_x^{|\vec{P}|}\,\vec{Q}\}\{\Uparrow_\kappa^{|\vec{N}|}\Uparrow_x^{|\vec{P}|}\,\vec{R}\})\{\vec{N}\}\{\vec{P}\} \;=\; M\{\vec{N},\vec{Q}\}\{\vec{P},\vec{R}\}$

4 Verification of a Non-optimizing CPS Transformation

The non-optimizing CPS transformation for our source language is a straightforward extension of Plotkin's original call-by-value CPS transformation. We define two mutually recursive transformations, Ψ for atoms and $[\![\cdot]\!]$ for arbitrary terms. Atoms are defined by the following grammar:

Atoms: $\quad A ::= x_n \mid \lambda^n.\ M \mid \mu^n.\ M \mid C(A_1, \ldots, A_n)$

The transformation is defined by the following equations:

$$
\begin{aligned}
\Psi(x_n) &= x_n \\
\Psi(\lambda^n.\ M) &= \lambda^{n+1}.\ [\![M]\!](\kappa_0) \\
\Psi(\mu^n.\ M) &= \mu^{n+1}.\ [\![M]\!](\kappa_0) \\
\Psi(C(A_1, \ldots, A_n)) &= C(\Psi(A_1), \ldots, \Psi(A_n))
\end{aligned}
$$

$$
\begin{aligned}
[\![A]\!] &= \lambda^0.\ \kappa_0(\Psi(A)) \\
[\![M(N_1, \ldots, N_n)]\!] &= \lambda^0.\ [\![M.N_1 \ldots N_n \ \texttt{then}\ \kappa_n(\kappa_{n+1}, \kappa_{n-1}, \ldots, \kappa_0)]\!] \\
[\![\texttt{let}\ M\ \texttt{in}\ N]\!] &= \lambda^0.\ [\![M]\!](\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![N]\!](\kappa_1)) \\
[\![C(N_1, \ldots, N_n)]\!] &= \lambda^0.\ [\![N_1 \ldots N_n \ \texttt{then}\ \kappa_n(C(\kappa_{n-1}, \ldots, \kappa_0)]\!] \\
&\qquad \text{if } C(N_1, \ldots, N_n) \text{ is not an atom} \\
[\![\texttt{match}\ M\ \texttt{with}\ \pi_1, \ldots, \pi_n]\!] &= \lambda^0.\ [\![M]\!](\lambda^0.\ \texttt{match}\ \kappa_0\ \texttt{with}\ [\![\pi_1]\!], \ldots, [\![\pi_n]\!]) \\
[\![M_1 \ldots M_n \ \texttt{then}\ N']\!] &= [\![M_1]\!](\lambda^0. \ldots [\![M_n]\!](\lambda^0.\ N') \ldots) \\
[\![C^n \to M]\!] &= C^n \to [\![M]\!](\kappa_1)
\end{aligned}
$$

The translation $[\![M]\!]$ of a source term M is always a one-argument abstraction $\lambda^0 \ldots$ that will receive the current continuation and bind it to variable κ_0. A source function of arity $n+1$ becomes a function of arity $n+2$ that expects the continuation of the call as first argument (bound to variable κ_0), along with $n+1$ regular arguments (bound to variables $x_n, \ldots, x_0$). For n-ary applications of functions and constructors, we use an auxiliary transformation for lists of expressions, written $[\![M_1 \ldots M_n \ \texttt{then}\ N']\!]$. The generated term evaluates the translations $[\![M_1]\!], \ldots, [\![M_n]\!]$ and binds them to $\kappa_{n-1}, \ldots, \kappa_0$ (respectively) before evaluating N'. In the case of a function application $M(N_1, \ldots, N_n)$, we translate the list $M.N_1 \ldots N_n$ and finish with κ_n bound to the translation of M, $\kappa_{n-1}, \ldots, \kappa_0$ bound to the translations of $N_1, \ldots, N_n$, and κ_{n+1} bound to the

outer continuation for the application. We therefore finish the computation by evaluating $\kappa_n(\kappa_{n+1}, \kappa_{n-1}, \ldots, \kappa_0)$.

The case of a constructor application is similar. However, if all arguments to the constructor are atoms, the constructor application itself is an atom and we force it to be translated as such. This not only improves the efficiency of the generated CPS term, but more importantly this is necessary for the proof of correctness to go through.

The CPS transformation satisfies the following syntactic properties, which play a crucial role in the proof of semantic preservation. We say that a term is κ-closed if no κ_i variables appear free in this term.

Lemma 3. *$[\![M]\!]$ and $\Psi(A)$ are κ-closed. As a corollary, transformed terms are invariant by substitution of κ-variables:*

$$[\![M]\!]\{\vec{N}\}\{\vec{P}\} = [\![M]\!]\{\,\}\{\vec{P}\}$$

Proof. By structural induction over M and A. For the n-ary applications, notice that κ_i is free in $[\![M_1 \ldots M_n \ \texttt{then} \ N']\!]$ only if κ_{i+n} is free in N'.

Lemma 4. *The transformation commutes with substitution of atoms for x-variables:*

$$[\![M\{A_1, \ldots, A_n\}]\!] = [\![M]\!]\{\,\}\{\Psi(A_1) \ldots \Psi(A_n)\}$$
$$\Psi(A\{A_1, \ldots, A_n\}) = \Psi(A)\{\,\}\{\Psi(A_1) \ldots \Psi(A_n)\}$$

Proof. By structural induction over M and A. Notice that atoms are stable by substitution: $A\{A_1, \ldots, A_n\}$ is an atom whenever $A, A_1, \ldots, A_n$ are atoms.

To show that the CPS transformation preserves the semantics of the source program, we would like to show that if the source program P evaluates to the value v, then the CPS program $[\![P]\!]$ applied to the initial continuation $\lambda^0.\ \kappa_0$ (the identity function) evaluates to the value $\Psi(v)$, which has the same shape as v and differs only on the bodies of functions contained in v. Of course, this result cannot be proved by induction over P: we need to generalize the result to continuations other than the initial continuation.

The intuition for this generalization is simple: if $M \Rightarrow v$, the intended effect for the transformation $[\![M]\!]$ applied to a continuation K is to compute the value $\Psi(v)$, then apply K to this value. Therefore, whenever $K\ \Psi(v) \Rightarrow v'$, it should be the case that $[\![M]\!](K) \Rightarrow v'$.

Lemma 5. *Let $K = \lambda^0.P$ be a κ-closed, one-argument abstraction of the target language. If $M \Rightarrow v$ in the source language, and $P\{\Psi(v)\}\{\,\} \Rightarrow v'$ in the target language, then $[\![M]\!](K) \Rightarrow v'$ in the target language.*

Proof. The proof proceeds by induction on the evaluation derivation of $M \Rightarrow v$ and case analysis over the term M. To give an idea of the proof, we sketch one case of intermediate difficulty: the case where $M = \texttt{let}\ M_1\ \texttt{in}\ M_2$. We have $M_1 \Rightarrow v_1$ and $M_2\{v_1\} \Rightarrow v$. We need to show

$$(\lambda^0.\ [\![M_1]\!](\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\kappa_1)))\ (K) \Rightarrow v' \tag{1}$$

under the assumptions that $K = \lambda^0.P$, K is κ-closed, and $P\{\Psi(v)\}\{\,\} \Rightarrow v'$.

Applying the induction hypothesis to the second premise $M_2\{v_1\} \Rightarrow v$ and the continuation K, we obtain:

$$[\![M_2\{v_1\}]\!](K) \Rightarrow v' \tag{2}$$

By Lemma 4 and the fact that v_1 is a value and therefore also an atom, (2) is equivalent to:

$$([\![M_2]\!]\{\,\}\{\Psi(v_1)\})(K) \Rightarrow v' \tag{3}$$

Take $P_1 = \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\Uparrow^1_x K)$. By the evaluation rule for `let`, Lemma 3, and some calculation over substitutions, (3) implies

$$P_1\{\Psi(v_1)\}\{\,\} \Rightarrow v' \tag{4}$$

The expected result (1) follows from (4) and the induction hypothesis applied to the first premise $M_1 \Rightarrow v_1$ and to the continuation $K_1 = \lambda^0.P_1$, which is κ-closed by Lemma 3.

Theorem 1. *If $M \Rightarrow v$ in the source language, then $[\![M]\!](\lambda^0.\ \kappa_0) \Rightarrow \Psi(v)$ in the target language. Moreover, if v is a first-order data structure (composed of constructors, but containing no function abstractions), then $[\![M]\!](\lambda^0.\ \kappa_0) \Rightarrow v$ in the target language.*

Proof. We apply Lemma 5 to the initial continuation $K = \lambda^0.\ \kappa_0$, obtaining $[\![M]\!](\lambda^0.\ \kappa_0) \Rightarrow \Psi(v)$. For the corollary, we observe that $\Psi(v) = v$ for any first-order data structure v.

5 Verification of an Optimizing CPS Transformation

We now define an optimized CPS transformation that does not generate administrative redexes. This transformation generalizes transformation $[\![\cdot]\!]_4\ \triangleright \cdot$ from Sect. 2, namely the transformation of Danvy and Nielsen [8, 9] presented using a "smart application" constructor $@_\beta$. This constructor is defined over terms of the target language by

$$(\lambda^0.M)\ @_\beta\ A = M\{A\}\{\,\} \qquad\qquad M\ @_\beta\ N = M(N) \text{ otherwise}$$

The optimizing transformation is presented as two mutually recursive functions, a one-place function Ψ for atoms and a two-place function $[\![\cdot]\!]\ \triangleright \cdot$ for arbitrary terms.

$$\begin{aligned}
\Psi(x_n) &= x_n \\
\Psi(\lambda^n.\ M) &= \lambda^{n+1}.\ [\![M]\!] \triangleright \kappa_0 \\
\Psi(\mu^n.\ M) &= \mu^{n+1}.\ [\![M]\!] \triangleright \kappa_0 \\
\Psi(C(A_1, \ldots, A_n)) &= C(\Psi(A_1), \ldots, \Psi(A_n))
\end{aligned}$$

$$
\begin{aligned}
[\![A]\!] \rhd k &= k \mathbin{@_\beta} \Psi(A) \\
[\![M(N_1, \ldots, N_n)]\!] \rhd k &= [\![M.N_1 \ldots N_n \ \texttt{then}\ \kappa_n(\Uparrow_\kappa^{n+1} k, \kappa_{n-1}, \ldots, \kappa_0)]\!] \\
[\![\texttt{let}\ M\ \texttt{in}\ N]\!] \rhd k &= [\![M]\!] \rhd \lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![N]\!] \rhd \Uparrow_\kappa^1 \Uparrow_x^1 k \\
[\![C(N_1, \ldots, N_n)]\!] \rhd k &= [\![N_1 \ldots N_n \ \texttt{then}\ \ \Uparrow_\kappa^n k(C(\kappa_{n-1}, \ldots, \kappa_0)]\!] \\
&\qquad \text{if } C(N_1, \ldots, N_n) \text{ is not an atom} \\
[\![\texttt{match}\ M\ \texttt{with}\ \pi_1, \ldots, \pi_n]\!] \rhd k &= [\![M]\!] \rhd \lambda^0.\ \texttt{match}\ \kappa_0\ \texttt{with} \\
&\qquad\qquad [\![\pi_1]\!] \rhd k, \ldots, [\![\pi_n]\!] \rhd k \\
[\![M_1 \ldots M_n \ \texttt{then}\ N']\!] &= [\![M_1]\!] \rhd \lambda^0. \ldots [\![M_n]\!] \rhd \lambda^0.\ N' \\
[\![C^n \to M]\!] \rhd k &= C^n \to [\![M]\!] \rhd \Uparrow_\kappa^1 k
\end{aligned}
$$

To show that the optimizing CPS transformation preserves semantics, we would like to prove an analogue of Theorem 1: if $M \Rightarrow v$ and v is a first-order data structure, then $[\![M]\!] \rhd \lambda^0.\ \kappa_0 \Rightarrow v$. However, a direct proof of this theorem in the style of Lemma 5 is difficult. The root of the problem is that the "smart application" $@_\beta$ does not commute with substitutions. For example,

$$(x_0 \mathbin{@_\beta} C)\{x_0 \leftarrow \lambda^0.\ \kappa_0\} \ = \ (x_0(C))\{x_0 \leftarrow \lambda^0.\ \kappa_0\} \ = \ (\lambda^0.\ \kappa_0)(C)$$

while

$$(x_0\{x_0 \leftarrow \lambda^0.\ \kappa_0\}) \mathbin{@_\beta} (C\{x_0 \leftarrow \lambda^0.\ \kappa_0\}) \ = \ (\lambda^0.\ \kappa_0) \mathbin{@_\beta} C \ = \ C$$

Consequently, the optimizing transformation does not commute with substitutions of atoms for x-variables, as was the case for the non-optimizing transformation (Lemma 4).

To avoid these difficulties, we do not attempt to directly prove the correctness of the optimizing transformations, but instead show a semantic equivalence result between the naive and the optimizing transformations. This equivalence builds on the intuition that $[\![M]\!] \rhd k$ is identical to $[\![M]\!](k)$ modulo the contraction of some administrative redexes. These contractions are instances of β_v reductions:

$$(\lambda^0.\ M)(A) \to M\{A\}\{\,\} \qquad (\beta_v)$$

where the argument A must be an atom. It is well known that β_v reductions are valid in call-by-value semantics [22].

We first formally define parallel β_v reduction between terms of the target language. This parallel reduction relation, written $\leadsto$, is defined by the inference rules in Fig. 3. The first rule corresponds to one β_v reduction. The other rules build the congruence closure of this reduction, enabling zero, one or several β_v redexes to be reduced simultaneously at any position in the term. The reflexive transitive closure of $\leadsto$ is written $\overset{*}{\leadsto}$.

We then formalize the intuition that the term $[\![M]\!] \rhd K$ can be obtained by contracting β_v redexes in the term $[\![M]\!](K)$.

Lemma 6. *For all atoms A, $\Psi(A) \overset{*}{\leadsto} \Psi(A)$. For all terms M and continuations K_1 and K_2, if $K_1 \overset{*}{\leadsto} K_2$ and K_1 is an atom, then $[\![M]\!]K_1 \overset{*}{\leadsto} [\![M]\!] \rhd K_2$.*

$$\frac{A_1 \rightsquigarrow A_2 \quad A_1 \text{ is an atom} \quad M \rightsquigarrow N}{(\lambda^0.\ M)\ A_1 \rightsquigarrow N\{A_2\}\{\,\}}$$

$$x_i \rightsquigarrow x_i \qquad \kappa_i \rightsquigarrow \kappa_i \qquad \frac{M \rightsquigarrow N}{\lambda^n.\ M \rightsquigarrow \lambda^n.\ N} \qquad \frac{M \rightsquigarrow N}{\mu^n.\ M \rightsquigarrow \mu^n.\ N}$$

$$\frac{M \rightsquigarrow N \quad M_1 \rightsquigarrow N_1 \quad \ldots \quad M_n \rightsquigarrow N_n}{M(M_1, \ldots, M_n) \rightsquigarrow N(N_1, \ldots, N_n)} \qquad \frac{M_1 \rightsquigarrow N_1 \quad M_2 \rightsquigarrow N_2}{(\texttt{let}\ M_1\ \texttt{in}\ M_2) \rightsquigarrow (\texttt{let}\ N_1\ \texttt{in}\ N_2)}$$

$$\frac{M_1 \rightsquigarrow N_1 \quad \ldots \quad M_n \rightsquigarrow N_n}{C(M_1, \ldots, M_n) \rightsquigarrow C(N_1, \ldots, N_n)}$$

$$\frac{M \rightsquigarrow N \quad \pi_1 \rightsquigarrow \pi_1' \quad \ldots \quad \pi_n \rightsquigarrow \pi_n'}{(\texttt{match M with}\ \pi_1 \ldots \pi_n) \rightsquigarrow (\ \texttt{match M with}\ \pi_1' \ldots \pi_n')} \qquad \frac{M \rightsquigarrow N}{C^n \to M \rightsquigarrow C^n \to N}$$

Fig. 3. Definition of the parallel β_v reduction $\rightsquigarrow$

Proof. By structural induction over A and M.

We then show that the $\rightsquigarrow$ relation preserves semantics, in the following sense:

Lemma 7. *If $M \Rightarrow v$ and $M \rightsquigarrow N$, then there exists a value w such that $N \Rightarrow w$ and $v \rightsquigarrow w$.*

Proof. By induction on the derivation of $M \Rightarrow v$. We use the following substitution lemma: if $M_1 \rightsquigarrow M_2$, $\vec{N} \rightsquigarrow \vec{Q}$ and $\vec{P} \rightsquigarrow \vec{R}$ where $\vec{N}$ and $\vec{P}$ are lists of values, then $M_1\{\vec{N}\}\{\vec{P}\} \rightsquigarrow M_2\{\vec{Q}\}\{\vec{R}\}$.

Combining these results, we obtain the correctness of the optimizing CPS transformation.

Theorem 2. *If $M \Rightarrow v$ in the source language, there exists a value w such that $\Psi(v) \overset{*}{\rightsquigarrow} w$ and $[\![M]\!] \rhd \lambda^0.\ \kappa_0 \Rightarrow w$ in the target language. Moreover, if v is a first-order data structure, then $[\![M]\!] \rhd \lambda^0.\ \kappa_0 \Rightarrow v$ in the target language.*

Proof. By Theorem 1, we know that $[\![M]\!](\lambda^0.\ \kappa_0) \Rightarrow \Psi(v)$. Lemma 6 shows that $[\![M]\!](\lambda^0.\ \kappa_0) \overset{*}{\rightsquigarrow} [\![M]\!] \rhd \lambda^0.\ \kappa_0$. Applying Lemma 7 repeatedly, we obtain the desired value w. If, moreover, v is a first-order data structure, then $\Psi(v) = v$, and $v \overset{*}{\rightsquigarrow} w$ implies $w = v$ by definition of the $\rightsquigarrow$ relation.

6 The Coq Development

The Coq mechanization of the results presented here is mostly standard. The CPS transformations, as well as the substitution and lifting operations, are presented as structural recursive functions. An advantage of this style is that Coq's extraction mechanism can generate executable Caml code directly from these

functional specifications — there is no need to manually implement these functions in a programming language. Coq puts strong syntactic restrictions on recursive functions to ensure that they always terminate. As presented in Sect. 4 and 5, our transformations violate these restrictions; we had to locally expand the transformations of values and lists at point of use in the transformations of general terms. The operational semantics for the languages are presented as inductive predicates where each constructor corresponds exactly to one inference rule in Fig. 1 and 2.

Concerning the integration of the CPS transformations into the mini-ML to Cminor compiler that we are developing and verifying, only the optimizing CPS transformation of Sect. 5 is actually used in the compiler. The naive transformation of Sect. 4 appears only as an intermediate step in its proof of correctness.

In the compiler chain, CPS transformation comes after an uncurrying optimization (described in [10]) and before closure conversion. The output language of the uncurrying pass, and the input language of the closure conversion pass, are identical to the source language of the CPS transformations as defined in Sect. 3. However, the correctness proofs of these passes are conducted against a big-step semantics for this language that uses environments and closures instead of simultaneous substitutions. To resolve these mismatches between the CPS transformation and the surrounding passes, we also formalized and proved correct a translation from the CPS target language (with two kinds of de Bruijn indices) back to the CPS source language (with a single kind of indices), as well as a semantic equivalence result between substitution-based and environment-based semantics.

The whole development took about 4 person.months and represents approximately 9000 lines of Coq, decomposed as follows:

	Specifications	Proofs
Languages and their semantics (Sect. 3)	624 lines	—
Substitutions and their properties	447 lines	1 685 lines
Non-optimizing CPS transformation (Sect. 4)	609 lines	1 676 lines
Parallel β_v reductions (Sect. 5)	413 lines	689 lines
Optimizing CPS transformation (Sect. 5)	113 lines	237 lines
Connecting uncurrying with CPS transformation	303 lines	516 lines
Connecting CPS transformation with closure conversion	644 lines	803 lines

Among the specifications, only 300 lines correspond to definitions of executable functions which will be integrated into the compiler itself after extraction.

As usual with mechanizations using de Bruijn indices, the definitions of substitution and lifting plus the proofs of their properties take up a large part of our development. Finding the correct statements of these properties is now well understood in the case of elementary substitutions, but required some trial and error in the case of simultaneous substitutions. The theory of the $\lambda\sigma$-calculus [1] helped us find the correct statements before attempting to prove them. Their proofs are large but mostly routine. We were able to partially automate these proofs using special-purpose tactics defined within Coq's `ltac` language.

7 Conclusions

The work presented in this paper shows that an optimizing CPS transformation defined for a realistic core functional language can be, with some effort, mechanically proved correct using a proof assistant. Our mechanization uses only elementary techniques (no higher-order abstract syntax, no nominal logic) and should therefore be adaptable to most proof assistants. We used several non-standard technical devices: de Bruijn notation with two kinds of indices; proving the CPS transformation against big-step operational semantics instead of small-step semantics; and proving the optimizing transformation by reduction to the non-optimizing one. These devices do not significantly reduce the overall size of the proof, but enable us to decompose it into mostly-independent sub-proofs of more manageable size. For instance, using two kinds of indices requires an additional transformation and a separate correctness proof for it, but minimizes the amount of index management performed during CPS transformation and keeps its correctness proof simple.

A natural extension of this work is to mechanically verify the correctness of transformations to A-normal forms [12] and monadic normal forms [23], two intermediate representations that share many of the features of CPS. We have not attempted to do so, but believe that the techniques presented here could be effective in these other settings.

Although the intended use for our mini-ML compiler is to compile strongly normalizing programs, it would be interesting to try to prove the correctness of CPS transformations for diverging programs using the co-inductive big-step semantics of [17].

Another direction for further work is to investigate the usability of Urban's Isabelle/HOL implementation of nominal logic [25] for proving the correctness of CPS transformations.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.
3. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.
5. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

6. A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Programming Language Design and Implementation 2007*, pages 54–65. ACM Press, 2007.
7. Coq development team. The Coq proof assistant. Software and documentation available at `http://coq.inria.fr/`, 1989–2007.
8. O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Computer Science*, 308(1-3):239–257, 2003.
9. O. Danvy and L. R. Nielsen. CPS transformation of beta-redexes. *Information Processing Letters*, 94(5):217–224, 2005.
10. Z. Dargaye. Décurryfication certifiée. In *Journées Francophones des Langages Applicatifs (JFLA'07)*. INRIA, 2007.
11. N. G. de Bruijn. 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, 1972.
12. C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation 1993*, pages 237–247. ACM Press, 1993.
13. A. Kennedy. Compiling with continuations, continued. In *International Conference on Functional Programming 2007*. ACM Press, 2007.
14. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
15. D. Kranz, N. Adams, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. ORBIT: an optimizing compiler for Scheme. In *SIGPLAN '86 symposium on Compiler Construction*, pages 219–233. ACM Press, 1986.
16. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE Computer Society Press, 2005.
17. X. Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 2006.
18. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
19. P. Letouzey. A new extraction for Coq. In *Types for Proofs and Programs, Workshop TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer-Verlag, 2003.
20. Y. Minamide and K. Okuma. Verifying CPS transformations in Isabelle/HOL. In *MERLIN '03: Proc. workshop on Mechanized reasoning about languages with variable binding*, pages 1–8. ACM Press, 2003.
21. J. S. Moore. *Piton: a mechanically verified assembly-language*. Kluwer, 1996.
22. G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
23. A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997.
24. Y. H. Tian. Mechanically verifying correctness of CPS compilation. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theory Symposium*, pages 41–51. Australian Computer Society, 2006.
25. C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 2007. To appear.