Verifying a CPS Transformation

Martijn Vermaat

mvermaat@cs.vu.nl

Bachelor Project, February 2008

Abstract

This paper is a case study on the mechanical verification of a transformation to continuation-passing style (CPS) by Dargaye and Leroy. First, the concepts of CPS and transformation to CPS are introduced. We then define source and target languages, and a CPS transformation between them which is proved to preserve semantics. We conclude with a small overview of the mechanization of CPS transformations. A detailed proof of the semantics preservation theorem is attached as an appendix.

1 Introduction

Continuation-passing style (CPS) is a functional programming style in which control is passed explicitely to every function in the form of a continuation. Functions written in continuation-passing style, as opposed to functions written in direct style, never return the result of their computation directly but instead pass it to a continuation function. This continuation function is received as an extra argument. For example, the unary function that returns its argument substracted by 2 is written $\lambda x.\, x - 2$ in direct style λ-calculus. Written in continuation-passing style, this function becomes $\lambda x.\, \lambda k.\, k\,(x - 2)$ where k is the continuation function.

A main property of programs written in CPS is that their evaluation is independent of the evaluation strategy. Evaluating a CPS program with a call-by-value strategy yields the same evaluation steps as evaluating it with a call-by-need strategy [14]. This makes CPS programs attractive as a semantical target.

In the field of compiler construction, CPS is often used as an intermediate language in compiling functional languages [1, 9]. Many optimizing transformations, such as inlining and memoization, are easier to perform on CPS programs than on direct-style programs.

Programs written in direct style can be mechanically transformed to equivalent programs in CPS [5]. Many of such transformations are known, the earliest of which is due to Plotkin [14]. A major disadvantage of Plotkin's original transformation is that it generates a lot of administrative redexes. While theoretically harmless, in practice these redexes must be removed in a later pass of an optimizig compiler and make proving certain properties of the transformation less straightforward. Therefore, a number of alternative transformations

to CPS have been proposed, some of them yielding no administrative redexes at all. We will see two examples of CPS transformations in Section 2.

Formal verification of programs is used to prove that a program does not have certain defects or has certain properties. Traditionally, program verification is done by proving the source code of the program correct. Afterwards, the source code is compiled to binary code which is then executed. It should be clear that this scheme does not provide any guarantees on the correctness of the executed code, one of the reasons being that the compiler could have altered the meaning of the program.

There are a number of ways to remedy this situation. The most obvious, and the one we will focus on, being to verify the correctness of the compiler before compiling the source code of the program. Other ways are verifying the result of the compilation (translation validation) [15, 13] and using proof-carrying code. While these alternatives do not require proving correctness of the compiler, this requirement is shifted to the validator and the proof-checker, respectively.

Dargaye and Leroy [7] have implemented and verified two CPS transformations using the Coq proof assistant [8, 3]. Their work is part of the larger project to implement a verified compiler for the mini-ML language, a restricted part of the ML functional language rich enough to be used as a target language for automatic extraction of programs from Coq specifications [11].

We will do a simplified case study on the setting of Dargaye and Leroy. Where their transformations work on a realistic functional source language featuring n-ary functions, recursive functions, algebraic data types and pattern-matching, we restrict this list to n-ary functions only. Furthermore, we focus on their first transformation, a straightforward extension of Plotkin's original transformation, ignoring their verification of an optimizing transformation. Finally, the mechanization of the transformation and its verification will not be discussed.

Contributions of this text include a more detailed description of the correctness proof of Plotkin's extended transformation and, hopefully, a simpler and more approachable presentation of the matter at hand.

2 CPS Transformations

Let us now look at two transformations from programs in direct style to programs in continuation-passing style. Plotkin's original call-by-value transformation on terms in the untyped λ-calculus is defined as follows:

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

An entire program M is now translated, using the initial continuation $\lambda x.\, x$ (the identity function), as $[\![M]\!]\, (\lambda x.\, x)$.

This transformation is conceptually easy, but unfortunately yields a lot of administrative redexes. An administrative redex is a redex introduced by the CPS transformation, thus not present in the original source term. Consider for example the transformation of $(\lambda x.\, x)\, y$ and subsequent β-reduction of several administrative redexes:

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

Note that the term concluding the above reduction still contains a redex, but this redex was already present in the source term. Administrative redexes are to be avoided, because they complicate formal reasoning about a transformation and add computation steps to a compilation process.

The following variation on Plotkin's transformation generates fewer administrative redexes. Instead of abstracting over the continuation function, the continuation is turned into an additional argument of the transformation which is thus defined as a binary function $[\![M]\!] \triangleright k$ on a source term M and a continuation term k:

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

where the complete translation of a program M becomes $[\![M]\!] \triangleright (\lambda x.\, x)$.

Considering the term $(\lambda x.\, x)\, y$ again, transforming it will now generate fewer administrative redexes:

$$\begin{aligned}
[\![(\lambda x.\, x)\, y]\!] \triangleright k &= (\lambda m.\, (\lambda n.\, m\, n\, k)\, y)\, (\lambda x\, k.\, k\, x) \\
&\to_\beta (\lambda n.\, (\lambda x\, k.\, k\, x)\, n\, k)\, y \\
&\to_\beta (\lambda x\, k.\, k\, x)\, y\, k
\end{aligned}$$

These are just two of many known CPS transformations. For example, a transformation that avoids generating any administrative redexes at all is described by Danvy and Nielsen [6]. In this text, we will consider an extension of the original transformation by Plotkin.

3 Languages

We will now describe the syntax and semantics of two languages acting as the source and target languages of the transformation. They are direct simplifications of the languages used in [7], not containing recursive functions, algebraic datatypes, and pattern matching. The result is two simple extensions of untyped λ-calculus with n-ary functions and `let`-bindings.

In order to avoid difficulties with α-conversion, de Bruijn indices will be used instead of named variables. In this notation, variables are indexed by a natural number denoting the number of abstractions in scope between the variable and its binding abstraction. Now α-equivalence of terms simply corresponds to syntactic equality. As an example, consider the term $\lambda x.\lambda y.\lambda z.x\,z\,(y\,z)$ in the λ-calculus with named variables. The same term written with de Bruijn indexed variables becomes $\lambda\,\lambda\,\lambda\,x_2\,x_0\,(x_1\,x_0)$.

The source and target languages differ only in that the target language features two kinds of variables (independently indexed), the extra kind being used for continuations and intermediate evaluation results introduced by the CPS transformation.

3.1 Source Language

Terms M in the source language are defined by the following grammar:

$M ::= x_n$		variable
	$\mid \lambda^n.M$	abstraction of arity $n+1$
	$\mid M(M,\ldots,M)$	function application
	$\mid$ `let` M `in` M	binding

where n ranges over the natural numbers.

Variables x_i are identified by their de Bruijn indices i. An abstraction $\lambda^n.M$ binds variables $x_n,\ldots,x_0$ in M. The `let`-binding `let` M `in` N binds x_0 to M in N.

Simultaneous substitution of terms $N_0,\ldots,N_n$ for $x_0,\ldots,x_n$ in M is written $M\{N_0,\ldots,N_n\}$. Lifting of free variables is denoted by the lifting operator: $\Uparrow^n M$ replaces all free x_i in M by x_{i+n}. A value v is any term of the form $\lambda^n.M$.

The big-step operational semantics of the source language is given in Figure 1. The three rules define the evaluation relation $M \Rightarrow v$ on terms M and values v, reading "M evaluates to v". As can be seen from the `let`-rule and the rule for function application, arguments are evaluated first, making this a call-by-value semantics.

An example term is $\lambda^1.\,x_0\,(x_1, (\lambda^0.\,x_0))$, a function expecting two arguments and returning the second argument applied to the first argument and the identity function.

3.2 Target Language

The target language is a direct adaptation of the source language, adding continuation variables κ_i (we will refer to variables x_i as source-level variables). The two types of variables are numbered independently by their de Bruijn indices.

$$\textsc{Values}\ \frac{}{\lambda^n.M \Rightarrow \lambda^n.M} \qquad \texttt{let}\ \frac{M \Rightarrow v_1 \qquad N\{v_1\} \Rightarrow v}{(\texttt{let}\ M\ \texttt{in}\ N) \Rightarrow v}$$

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

Figure 1: Source language semantics

$$\textsc{Values}\ \frac{}{\lambda^n.M' \Rightarrow \lambda^n.M'} \qquad \texttt{let}\ \frac{M' \Rightarrow v_1 \qquad N'\{\}\{v_1\} \Rightarrow v}{(\texttt{let}\ M'\ \texttt{in}\ N') \Rightarrow v}$$

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

Figure 2: Target language semantics

Terms M' in the target language are defined by the following grammar:

$M' ::= x_n$	source-level variable
$\mid\ \kappa_n$	continuation variable
$\mid\ \lambda^n.M'$	abstraction of arity $n+1$
$\mid\ M'(M', \ldots, M')$	function application
$\mid\ \texttt{let}\ M'\ \texttt{in}\ M'$	binding

where n ranges over the natural numbers.

An abstraction $\lambda^n.M'$ binds its first argument, the continuation, to κ_0, and its remaining arguments to $x_{n-1}, \ldots, x_0$ in M'. `let`-bindings behave analogous to the same construct in the source language.

The double simultaneous substitution of terms $N'_0, \ldots, N'_n$ for $\kappa_0, \ldots, \kappa_n$ and terms $P'_0, \ldots, P'_m$ for $x_0, \ldots, x_m$ in M' is written $M'\{N'_0, \ldots, N'_n\}\{P'_0, \ldots, P'_m\}$. We will use the following notation to distinguish between lifting of source-level variables and lifting of continuation variables: $\Uparrow_x^n M$ lifts free x_i by n in M and $\Uparrow_\kappa^n M$ lifts free κ_i by n in M.

The big-step operational semantics for the target language is given in Figure 2.

4 Transformation

Again, we will follow the presentation of [7]. The transformation we will prove correct is a straightforward extension of Plotkin's call-by-value transformation, defined below by two mutually recursive functions Ψ on atoms and $[\![\cdot]\!]$ on terms.

An atom A is a variable or an abstraction:

$$A ::= \; x_n \mid \lambda^n.M$$

Note that $[\![\cdot]\!]$ really works on source language terms extended with a `then` construct used in the transformation of function applications.

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

$$\begin{aligned}
[\![A]\!] &= \lambda^0.\kappa_0(\Psi(A)) \\
[\![M(N_1,\ldots,N_n)]\!] &= \lambda^0.[\![M.N_1 \ldots N_n \texttt{ then } \kappa_n(\kappa_{n+1},\kappa_{n-1},\ldots,\kappa_0)]\!] \\
[\![\texttt{let } M \texttt{ in } N]\!] &= \lambda^0.[\![M]\!](\lambda^0.\texttt{let } \kappa_0 \texttt{ in } [\![N]\!](\kappa_1)) \\
[\![M_1 \ldots M_n \texttt{ then } N']\!] &= [\![M_1]\!](\lambda^0 \ldots [\![M_n]\!](\lambda^0.N') \ldots)
\end{aligned}$$

The transformation $[\![M]\!]$ of any source term M is a one-argument abstraction in the target language that will receive the current continuation and bind it to the continuation variable κ_0. A function of arity n in the source language is transformed to a function of arity $n+1$ in the target language expecting a continuation as its first argument (bound to κ_0) and n regular arguments thereafter (bound to $x_{n-1},\ldots,x_0$).

The transformation of the term $M_1 \ldots M_n \texttt{ then } N'$ first evaluates the terms $[\![M_1]\!],\ldots,[\![M_n]\!]$ and binds them to $\kappa_{n-1},\ldots,\kappa_0$ before evaluating N'. So in the transformation of the function application $M(N_1,\ldots,N_n)$, we have κ_n bound to $[\![M]\!]$, $\kappa_{n-1},\ldots,\kappa_0$ bound to $[\![N_1]\!],\ldots[\![N_n]\!]$, and κ_{n+1} bound to the outer continuation, after which $\kappa_n(\kappa_{n+1},\kappa_{n-1},\ldots,\kappa_0)$ is evaluated.

As an example, consider the transformation $[\![(\lambda^0.x_0)(\lambda^1.x_0)]\!]$, yielding

$$\begin{aligned}
&\lambda^0.\ \big(\lambda^0.\kappa_0(\lambda^1.\ (\lambda^0.\kappa_0(x_0))\ (\kappa_0)\ )\big) \\
&\quad \big(\lambda^0.\ (\lambda^0.\kappa_0(\lambda^2.(\lambda^0.\kappa_0(x_0))(\kappa_0)))\ (\lambda^0.\kappa_1(\kappa_2,\kappa_0))\ \big)\ .
\end{aligned}$$

We will say that a target language term is κ-closed if it does not contain any free continuation variables κ_i. The following lemmas state some important properties of the transformation. They will be proved in appendix A.

Lemma 1. *$[\![M]\!]$ and $\Psi(A)$ are κ-closed. As a corollary, transformed terms are invariant by substitution of κ-variables:*

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

Lemma 2. *The transformation commutes with substitution of atoms for source-level variables:*

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

5 Semantics Preservation

We will now show that the transformation $[\![\cdot]\!]$ from source language terms to target language terms preserves semantics. That is, if a term M in the source language evaluates to a value v, the translation $[\![M]\!]$ of M applied to the initial continuation $\lambda^0.\kappa_0$, should evaluate to the target language representation $\Psi(v)$ of v. This property is stated in Theorem 1.

Theorem 1. *If $M \Rightarrow v$ in the source language, then $[\![M]\!](\lambda^0.\kappa_0) \Rightarrow \Psi(v)$ in the target language.*

For a proof by induction to go through we need to generalize this result. The statement we will prove is given as Lemma 3 and generalizes the use of the initial continuation to any κ-closed, one-argument abstraction.

Lemma 3. *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. We will sketch the outline of a proof, a more elaborate version can be found in appendix A. The proof is by induction on the evaluation derivation of $M \Rightarrow v$ and case analysis over the term M.

Base case $M = \lambda^n.M_1$

We need to show $(\lambda^0.\kappa_0(\Psi(\lambda^n.M_1)))(K) \Rightarrow v'$. This is trivial.

Inductive case $M = M_1(N_0, \ldots, N_n)$

Premises are $M_1 \Rightarrow \lambda^n.Q$, $N_i \Rightarrow v_i$, and $Q\{v_n, \ldots, v_0\} \Rightarrow v$. Our goal is to show

$$[\![M_1]\!](\lambda^0.[\![N_0]\!](\lambda^0.\cdots[\![N_n]\!](\lambda^0.\kappa_{n+1}(K, \kappa_n, \ldots, \kappa_0))\cdots)) \Rightarrow v' \ .$$

We apply the induction hypothesis to the first premise and our new goal is

$$[\![N_0]\!](\lambda^0.\cdots[\![N_n]\!](\lambda^0.\Psi(\lambda^n.Q)(K, \kappa_n, \ldots, \kappa_0))\cdots) \Rightarrow v' \ .$$

Repeatedly applying the induction hypothesis to premises $N_i \Rightarrow v_i$ (starting with $i = 0$) leaves us to show

$$[\![Q\{v_n, \ldots, v_0\}]\!](K) \Rightarrow v' \ ,$$

which follows from the induction hypothesis applied to the last premise and the assumption $P\{\Psi(v)\}\{\} \Rightarrow v'$.

Inductive case $M = \texttt{let}\ M_1\ \texttt{in}\ M_2$

Premises are $M_1 \Rightarrow v_1$ and $M_2\{v_1\} \Rightarrow v$. The induction hypothesis applied to the second premise gives us

$$[\![M_2\{v_1\}]\!](\Uparrow^1_x \lambda^0.P) \Rightarrow v' \ .$$

Via the `let`-evaluation rule, we can now show

$$[\![M_1]\!](\lambda^0.\,\texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\Uparrow^1_x \lambda^0.P)) \Rightarrow v'$$

by applying the induction hypothesis to the first premise. The assumption $P\{\Psi(v)\}\{\} \Rightarrow v'$ and evaluation rule for function application complete the proof. □

Theorem 1 is an instantiation of Lemma 3, so we can now prove it easily.

Proof of Theorem 1. We take for K in Lemma 3 the initial continuation $\lambda^0.\kappa_0$, yielding $[\![M]\!](\lambda^0.\kappa_0) \Rightarrow \Psi(v)$. □

6 Mechanically Verified Proofs

Results such as that of the previous section can be mechanized using formal tools (e.g. a proof assistant). Reasons for doing so are several. Firstly, a mechanically checked proof is often more precise and convincing than a proof on paper. Secondly, mechanical proofs as are often easier to be reused in many ways, e.g. for composition, modification, and extension. Most importantly, the ultimate goal in the verification of these program transformations is of course obtaining correctness proofs of actual implementations (which are used directly in an application) and not just of transformations merely defined on paper.

However, compared to the traditional way of proving things, mechanical proofs come with a price. On paper, typically a lot of handwaving is done, leaving out a lot of the trivial details. Compare this to a formal system where one has to be precise, giving us confidence in the correctness of proofs on the one hand, but often making them tedious to write and hard to read on the other. In addition to this, reasoning mechanically about programs brings technical problems, such as formalizing the binding of variables, described in the POPLmark challenge [2]. Several approaches to these problems exist and we will discuss some of them below.

6.1 Mechanized Verification of CPS Transformations

The transformation described in Section 4 and an optimized transformation by Danvy and Nielsen [6] have been implemented and verified by Dargaye and Leroy [7] using the Coq proof assistant. The complete Coq development consists of 9000 lines of code. An implementation of the transformations in Caml is automatically extracted from the Coq specifications. The source and target languages are realistic functional languages, much like the ones introduced in Section 3, but extended with recursive functions, algebraic datatypes, and pattern matching. This is particularly original in that earlier work mostly considers the pure λ-calculus.

Binding of variables is represented using de Bruijn indices, hereby omitting the issue of α-equivalence between terms. Unfortunately, another complexity is introduced by the required lifting of indices in substitutions. A large part of

the Coq code deals with the properties of lifting and substitution. Furthermore, terms with de Bruijn variables are arguably harder to deal with for humans than terms with named variables.

To avoid much of the problems with lifting, two sorts of de Bruijn indices are used – one for source-level variables and one for continuation variables. Lifting of source-level variables is therefore no longer necessitated by the introduction of continuation variable bindings.

The transformations are verified using big-step operational semantics. While small-step semantics are more expressive (they can describe non-terminating computations), they tend to make program transformations significantly harder to prove correct. In this case, difficulties caused by administrative redexes are avoided by using big-step semantics. Furthermore, the intended use of the transformation is in a compiler for automatically extracted programs from Coq specifications, which are strongly normalizing.

6.2 Related Work

Other verified mechanizations of transformations to CPS differ on several aspects, most notably the logical framework used to implement the verification, the way languages and semantics are represented, and the type of operational semantics used.

An important decision for the representation of terms is how to formalize binding of variables. Named variables and α-conversion as we are used to on paper bring many difficulties to mechanical reasoning. A well-known and often used alternative are de Bruijn indices, although they bring the complexity of lifting. Higher-order abstract syntax is another frequently used technique, in which variables do not refer to their binder by name (i.e. as in first-order abstract syntax), but point directly to the binding site. The equivalence classes of Pitt's nominal logic provide us with yet another way to model α-equivalence of terms on a syntactical level, but are unfortunately not yet well supported in logical frameworks. Finally, a combination of nominal logic and de Bruijn indices is called a "locally nameless" representation [10], where free variables are named and bound variables are represented by their de Bruijn index.

A correctness proof of a transformation is based on some sort of semantics for the source and target languages. Usually, correctness is intepreted as preservation of semantics or observable behavior. A denotational semantics translates terms to mathematical structures in a compositional way. These mathematical structures, called denotations, describe the behavior of the language. An operational semantics gives meaning to a program by describing a computation. Two main styles of operational semantics exist. Traditional small-step semantics describe a series of computations formed by repeatedly applying one-step rules. Big-step semantics relate programs and their computational result directly.

Minamide and Okuma [12] verified several CPS transformations for λ-calculus in Isabelle/HOL. Terms are represented using first-order abstract syntax and named variables. This representation is choosen because it is both close to

what is used on paper and, so they argue, to intermediate languages used in compilers.

As noted above, a number of difficulties arise from the use of named variables, one of them being that of α-convertibility. This is tackled by reformulating α-equivalence as a syntax-directed deductive system. Other problems require that variables introduced by the transformation are fresh and that `let`-bound variables are unique.

Minamide and Okuma use a call-by-value operational semantics in traditional small-step style.

In [16], Tian implements and verifies a higher-order one-pass CPS transformation by Danvy and Nielsen in the logical framework Twelf. The source and target languages are formalized using higher-order abstract syntax and are accompanied by a big-step and small-step operational semantics, respectively.

Because the transformation is higher-order, it avoids problems with administrative redexes. The issues of binding and substitution are addressed by using higher-order abstract syntax.

Chlipala [4] used Coq to implement a certified compiler from simply typed λ-calculus to assembly language. The first two passes of the compiler implement a transformation to CPS.

Chlipala aims for as much automation in the proofs as possible. De Bruijn indices are used to represent variables and terms are given dependent types. Preservation of types is trivially proved by the dependent types and enables type-directed proof search to automate large parts of the semantics preservation proofs. The languages are accompanied by a type-directed denotational semantics.

Acknowledgements

Many thanks are due to Femke van Raamsdonk and Roel de Vrijer for their supervision. Your patience seemed unbounded but must have been tested to the limit. Heiltje Winterink provided comments on my English writing, thanks for that. And finally, *muchas gracias* Martina.

A Proofs

A.1 Lemma 1

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

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

Proof. The proof is by structural induction over M and case analysis over A.

Base case $M = x_i$

$[\![M]\!] = \lambda^0.\kappa_0(x_i)$, which is obviously κ-closed.

Inductive case $M = \lambda^n.M_1$

By the induction hypothesis, $[\![M_1]\!]$ is κ-closed. Therefore, also $[\![M]\!] = \lambda^0.\kappa_0(\lambda^{n+1}.[\![M_1]\!](\kappa_0))$ is κ-closed.

Inductive case $M = M_1(N_0, \ldots, N_n)$

$$\begin{aligned}[\![M]\!] &= \lambda^0.[\![M_1.N_0 \ldots N_n \texttt{ then } \kappa_{n+1}(\kappa_{n+2}, \kappa_n, \ldots, \kappa_0)]\!] \\ &= \lambda^0.[\![M_1]\!](\lambda^0.[\![N_0]\!](\lambda^0.\ldots.[\![N_n]\!](\lambda^0.\kappa_{n+1}(\kappa_{n+2}, \kappa_n, \ldots, \kappa_0))\cdots))\end{aligned}$$

By the induction hypothesis, $[\![M_1]\!]$, $[\![N_0]\!], \ldots, [\![N_n]\!]$ are κ-closed. There are $n+3$ abstractions surrounding the κ_{n+2}, so $[\![M]\!]$ is κ-closed.

Inductive case $M = \texttt{let } M_1 \texttt{ in } M_2$

$[\![M]\!] = \lambda^0.[\![M_1]\!](\lambda^0.\texttt{ let } \kappa_0 \texttt{ in } [\![M_2]\!](\kappa_1))$ is κ-free because $[\![M_1]\!]$ and $[\![M_2]\!]$ are (by the induction hypothesis).

Now, κ-freeness of $\Psi(A)$ and the corollary follow trivially. □

A.2 Lemma 2

The transformation commutes with substitution of atoms for source-level variables:

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

Proof. We omit lifting of continuation variables when there are none (implicitely applying Lemma 1). Furthermore, we use the fact $\Uparrow_x^n \Psi(A) = \Psi(\Uparrow^n A)$ several times. The first statement is proved by a structural induction argument over M.

Base case $M = x_i$

If $i < n$, we have

$$\begin{aligned}
[\![x_i\{A_1,\ldots,A_n\}]\!] &= [\![A_{i+1}]\!] \\
&= \lambda^0.\kappa_0(\Psi(A_{i+1})) \\
&= \lambda^0.\kappa_0(x_i)\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
&= [\![x_i]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\}\ .
\end{aligned}$$

Otherwise,

$$\begin{aligned}
[\![x_i\{A_1,\ldots,A_n\}]\!] &= [\![x_i]\!] \\
&= \lambda^0.\kappa_0(x_i) \\
&= \lambda^0.\kappa_0(x_i)\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
&= [\![x_i]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\}\ .
\end{aligned}$$

Inductive case $M = \lambda^m.M_1$

We use the induction hypothesis in the third step below:

$$\begin{aligned}
& [\![(\lambda^m.M_1)\{A_1,\ldots,A_n\}]\!] \\
= \ & [\![\lambda^m.M_1\{x_0,\ldots,x_m,\Uparrow^{m+1} A_1,\ldots,\Uparrow^{m+1} A_n\}]\!] \\
= \ & \lambda^0.\kappa_0(\lambda^{m+1}.[\![M_1\{x_0,\ldots,x_m,\Uparrow^{m+1} A_1,\ldots,\Uparrow^{m+1} A_n\}]\!](\kappa_0)) \\
= \ & \lambda^0.\kappa_0(\lambda^{m+1}.[\![M_1]\!] \\
& \qquad \{\}\{x_0,\ldots,x_m,\Uparrow_x^{m+1} \Psi(A_1),\ldots,\Uparrow_x^{m+1} \Psi(A_n)\}(\kappa_0)) \\
= \ & (\lambda^0.\kappa_0(\lambda^{m+1}.[\![M_1]\!](\kappa_0)))\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
= \ & [\![\lambda^m.M_1]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\}\ .
\end{aligned}$$

Inductive case $M = M_1(N_0,\ldots,N_m)$

In the fourth step below we use the induction hypothesis:

$$\begin{aligned}
& [\![M_1(N_0,\ldots,N_m)\{A_1,\ldots,A_n\}]\!] \\
= \ & [\![M_1\{A_1,\ldots,A_n\}(N_0\{A_1,\ldots,A_n\},\ldots,N_m\{A_1,\ldots,A_n\})]\!] \\
= \ & \lambda^0.[\![M_1\{A_1,\ldots,A_n\}.N_0\{A_1,\ldots,A_n\}\ldots N_m\{A_1,\ldots,A_n\} \\
& \qquad \texttt{then}\ \kappa_{m+1}(\kappa_{m+2},\kappa_m,\ldots,\kappa_0)]\!] \\
= \ & \lambda^0.[\![M_1\{A_1,\ldots,A_n\}]\!](\lambda^0.[\![N_0\{A_1,\ldots,A_n\}]\!] \\
& \qquad (\lambda^0.\ldots.[\![N_m\{A_1,\ldots,A_n\}]\!](\lambda^0.\kappa_{m+1}(\kappa_{m+2},\kappa_m,\ldots,\kappa_0))\cdots)) \\
= \ & \lambda^0.[\![M_1]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\}(\lambda^0.[\![N_0]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
& \qquad (\lambda^0.\ldots.[\![N_m]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
& \qquad\qquad (\lambda^0.\kappa_{m+1}(\kappa_{m+2},\kappa_m,\ldots,\kappa_0))\cdots)) \\
= \ & (\lambda^0.[\![M_1]\!](\lambda^0.[\![N_0]\!](\lambda^0.\ldots.[\![N_m]\!](\lambda^0.\kappa_{m+1}(\kappa_{m+2},\kappa_m,\ldots,\kappa_0))\cdots))) \\
& \qquad \{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
= \ & (\lambda^0.[\![M_1.N_0\ldots N_m\ \texttt{then}\ \kappa_{m+1}(\kappa_{m+2},\kappa_m,\ldots,\kappa_0)]\!]) \\
& \qquad \{\}\{\Psi(A_1),\ldots,\Psi(A_n)\} \\
= \ & [\![M_1(N_0,\ldots,N_m)]\!]\{\}\{\Psi(A_1),\ldots,\Psi(A_n)\}\ .
\end{aligned}$$

Inductive case $M = \texttt{let}\ M_1\ \texttt{in}\ M_2$

Here the induction hypothesis is used in the third step:

$$\begin{aligned}
& [\![(\texttt{let}\ M_1\ \texttt{in}\ M_2)\{A_1, \ldots, A_n\}]\!] \\
= \ & [\![\texttt{let}\ M_1\{A_1, \ldots, A_n\}\ \texttt{in}\ M_2\{x_0, \Uparrow^1 A_1, \ldots, \Uparrow^1 A_n\}]\!] \\
= \ & \lambda^0.[\![M_1\{A_1, \ldots, A_n\}]\!] \\
& \qquad (\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2\{x_0, \Uparrow^1 A_1, \ldots, \Uparrow^1 A_n\}]\!](\kappa_1)) \\
= \ & \lambda^0.[\![M_1]\!]\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\} \\
& \qquad (\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!]\{\}\{x_0, \Uparrow^1_x \Psi(A_1), \ldots, \Uparrow^1_x \Psi(A_n)\}(\kappa_1)) \\
= \ & \lambda^0.[\![M_1]\!](\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\kappa_1))\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\} \\
= \ & [\![\texttt{let}\ M_1\ \texttt{in}\ M_2]\!]\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\}\ .
\end{aligned}$$

A case analysis over A proves the second statement.

Base case $A = x_i$

If $i < n$, we have

$$\begin{aligned}
\Psi(x_i\{A_1, \ldots, A_n\}) &= \Psi(A_{i+1}) \\
&= x_i\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\} \\
&= \Psi(x_i)\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\}\ .
\end{aligned}$$

Otherwise,

$$\begin{aligned}
\Psi(x_i\{A_1, \ldots, A_n\}) &= \Psi(x_i) \\
&= x_i \\
&= x_i\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\} \\
&= \Psi(x_i)\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\}\ .
\end{aligned}$$

Inductive case $A = \lambda^m.M_1$

$$\begin{aligned}
& \Psi((\lambda^m.M_1)\{A_1, \ldots, A_n\}) \\
= \ & \Psi(\lambda^m.M_1\{x_0, \ldots, x_m, \Uparrow^{m+1} A_1, \ldots, \Uparrow^{m+1} A_n\}) \\
= \ & \lambda^{m+1}.[\![M_1\{x_0, \ldots, x_m, \Uparrow^{m+1} A_1, \ldots, \Uparrow^{m+1} A_n\}]\!](\kappa_0) \\
= \ & \lambda^{m+1}.[\![M_1]\!]\{\}\{\Uparrow^{m+1}_x \Psi(A_1), \ldots, \Uparrow^{m+1}_x \Psi(A_n)\}(\kappa_0) \\
= \ & (\lambda^{m+1}.[\![M_1]\!](\kappa_0))\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\} \\
= \ & \Psi(\lambda^m.M_1)\{\}\{\Psi(A_1), \ldots, \Psi(A_n)\}\ .
\end{aligned}$$

Here we used the first statement of this lemma, which we already proved above. □

A.3 Lemma 3

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. We assume $M \Rightarrow v$ and proceed by induction on its derivation, also assuming $P\{\Psi(v)\}\{\} \Rightarrow v'$.

Base case $M = \lambda^n.M_1$

This is the base case, where $v = \lambda^n.M_1$.

Because $[\![\lambda^n.M_1]\!] = \lambda^0.\kappa_0(\Psi(\lambda^n.M_1))$, we need to show

$$(\lambda^0.\kappa_0(\Psi(\lambda^n.M_1)))\,(K) \Rightarrow v' \ . \tag{1}$$

This is trivial, because according to the evaluation rule for function application we have

$$K\ (\Psi(\lambda^n.M_1)) \Rightarrow P\{\Psi(\lambda^n.M_1)\}\{\} \ .$$

Using the same evaluation rule and Lemma 1, we conclude (1).

Inductive case $M = M_1(N_0, \ldots, N_n)$

Premises of the evaluation rule for function application are $M_1 \Rightarrow \lambda^n.Q$, $N_i \Rightarrow v_i$, and $Q\{v_n, \ldots, v_0\} \Rightarrow v$.

Working out the transformation,

$$\begin{aligned}[\![M_1(N_0, \ldots, N_n)]\!] &= \lambda^0.[\![M_1.N_0 \ldots N_n \ \texttt{then}\ \kappa_{n+1}(\kappa_{n+2}, \kappa_n, \ldots, \kappa_0)]\!] \\ &= \lambda^0.[\![M_1]\!](\lambda^0.[\![N_0]\!](\lambda^0.\cdots \\ &\qquad \cdots [\![N_n]\!](\lambda^0.\kappa_{n+1}(\kappa_{n+2}, \kappa_n, \ldots, \kappa_0))\cdots)) \ ,\end{aligned}$$

we need to show

$$\begin{aligned}&(\lambda^0.[\![M_1]\!](\lambda^0.[\![N_0]\!](\lambda^0.\cdots \\ &\qquad \cdots [\![N_n]\!](\lambda^0.\kappa_{n+1}(\kappa_{n+2}, \kappa_n, \ldots, \kappa_0))\cdots)))(K) \Rightarrow v' \ .\end{aligned} \tag{2}$$

From the induction hypothesis applied to the premise $Q\{v_n, \ldots, v_0\} \Rightarrow v$ and continuation K we conclude

$$[\![Q\{v_n, \ldots, v_0\}]\!](K) \Rightarrow v' \ ,$$

and also, by Lemma 2 (v_i are atoms),

$$[\![Q]\!]\{\}\{\Psi(v_n), \ldots, \Psi(v_0)\}(K) \Rightarrow v' \ . \tag{3}$$

Furthermore, $\Psi(\lambda^n.Q) = \lambda^{n+1}.[\![Q]\!](\kappa_0)$, so the evaluation rule for function application gives us

$$\Psi(\lambda^n.Q)\ (K, \Psi(v_0), \ldots, \Psi(v_n)) \Rightarrow v' \tag{4}$$

by (3) via Lemma 1.

Let $P_1 = [\![N_0]\!](\lambda^0.\cdots[\![N_n]\!](\lambda^0.\kappa_{n+1}(K, \kappa_n, \ldots, \kappa_0))\cdots)$. We have

$$P_1\{\Psi(\lambda^n.Q)\}\{\} \Rightarrow v' \tag{5}$$

by (4) and repeatedly (for $i = 0 \ldots n$) applying the induction hypothesis to premises $N_i \Rightarrow v_i$ and continuations

$$\lambda^0.[\![N_{i+1}]\!](\lambda^0.\cdots[\![N_n]\!](\lambda^0.\kappa_{n+1}(K, \kappa_n, \ldots, \kappa_0))\cdots) \ .$$

Using this to apply the induction hypothesis to our first premise and continuation $\lambda^0.P_1$ (κ-closed by Lemma 1), we get

$$[\![M_1]\!](\lambda^0.P_1) \Rightarrow v' \ .$$

Now, (2) is proved using the evaluation rule for function application.

Inductive case $M = \texttt{let}\ M_1\ \texttt{in}\ M_2$

The evaluation rule for `let` gives us premises $M_1 \Rightarrow v_1$ and $M_2\{v_1\} \Rightarrow v$.

By definition of $[\![\cdot]\!]$, we have

$$[\![\texttt{let}\ M_1\ \texttt{in}\ M_2]\!] = \lambda^0.[\![M_1]\!](\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\kappa_1))$$

and thus need to show

$$(\lambda^0.[\![M_1]\!](\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\kappa_1)))(K) \Rightarrow v' \ . \tag{6}$$

Applying the induction hypothesis to the second premise and continuation K gives us $[\![M_2\{v_1\}]\!](K) \Rightarrow v'$, which is equivalent to

$$[\![M_2]\!]\{\}\{\Psi(v_1)\}(K) \Rightarrow v' \tag{7}$$

by Lemma 2 and the fact that v_1 is an atom.

Now consider the term $P_1 = \ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\Uparrow^1_x K)$. We will show

$$P_1\{\Psi(v_1)\}\{\} \Rightarrow v' \ . \tag{8}$$

By Lemma 1 there are no free continuation variables in $[\![M_2]\!](\Uparrow^1_x K)$, so

$$P_1\{\Psi(v_1)\}\{\} = \ \texttt{let}\ \Psi(v_1)\ \texttt{in}\ [\![M_2]\!](\Uparrow^1_x K) \ .$$

The `let`-evaluation rule states that (8) can now be concluded if $\Psi(v_1)$ evaluates to itself, which it does, and $([\![M_2]\!](\Uparrow^1_x K))\{\}\{\Psi(v_1)\} \Rightarrow v'$, which is equivalent to (7).

By (8), the induction hypothesis applied to the first premise and continuation $\lambda^0.P_1$ (κ-closed by Lemma 1) results in

$$[\![M_1]\!](\lambda^0.P_1) \Rightarrow v' \ .$$

From this, the expected result (6) follows via the evaluation rule for function application. □

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] B. E. Aydemir, A. Bohannon, M. Fairbairn, N. J. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2005.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] 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.

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

[6] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Computer Science*, 308(1-3):239–257, 2003.

[7] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2007)*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[8] Coq development team. The Coq proof assistant, 1989-2008. Software and documentation available at `http://coq.inria.fr/`.

[9] D. A. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. ORBIT: an optimizing compiler for Scheme. In *SIGPLAN Symposium on Compiler Construction*, pages 219–233, 1986.

[10] X. Leroy. A locally nameless solution to the POPLmark challenge. Technical report, INRIA, 2007.

[11] 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.

[12] Y. Minamide and K. Okuma. Verifying CPS transformations in Isabelle/HOL. In *MERLIN '03: Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pages 1–8. ACM Press, 2003.

[13] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[14] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[15] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK, 1998. Springer-Verlag.

[16] Y. H. Tian. Mechanically verifying correctness of CPS compilation. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 41–51. Australian Computer Society, 2006.