Under consideration for publication in J. Functional Programming

THEORETICAL PEARLS

Certification of higher-order one-pass CPS transformations

Małgorzata Biernacka
Institute of Computer Science
University of Wrocław
http://ii.uni.wroc.pl/~mabi
(*e-mail:* mabi@cs.uni.wroc.pl)

Abstract

We present a method for mechanically obtaining certified one-pass, higher-order transformations of lambda terms into continuation-passing style. Given a naive, non-optimizing encoding from the source to the target language, we adapt the proof of normalization à la Tait in the target language for a suitable axiomatization of administrative reductions; the computational content of the proof is an optimized transformation. We illustrate the applicability of this approach with three variants of the CPS transformation: eta-reduced call-by-value CPS, eta-expanded call-by-value CPS, and call-by-value CPS with generalized beta-reduction. In each of these cases, the resulting program coincides with a previously known transformation – we thus formally verify the correctness of these known transformations. We have also obtained one-pass transformations for the call-by-name counterparts of these transformations.

The development has been carried out in the Coq proof assistant and, by extraction from the proofs, we have obtained OCaml programs implementing the transformations. In addition, the factorization of the proof allows us to identify higher-order one-pass transformations as instances of normalization-by-evaluation programs fused with the naive translation.

1 Introduction

Translating lambda terms – and, more generally, functional programs – into continuation-passing style has proved to be a remarkably useful tool in both theory and practice of programming languages, originating from seminal work of Plotkin (Plotkin, 1975). Consequently, the topic has been extensively studied, and numerous variants of CPS transformations have been devised independently by various researchers and for different purposes. The task of devising one-pass CPS transformations has proven to be complex even though its extensional behavior can be characterized in a simple manner as the composition of Plotkin's non-optimizing (naive) translation (Plotkin, 1975) with subsequent normalization in the target code, i.e., reducing all the "administrative redices" introduced by the naive translation. The normalization phase should only reduce the administrative redices and not the ones present in the source term. This characterization has prompted various approaches to implement CPS translations efficiently in one pass. One approach is to use a higher-

order, compositional transformation and it has been proposed independently by Appel (Appel, 1992), by Danvy and Filinski (Danvy & Filinski, 1990; Danvy & Filinski, 1992), and by Wand (Wand, 1991). Another way is to use a first-order transformation, either context-based and non-compositional (Sabry & Felleisen, 1993), or compositional (Danvy & Nielsen, 2003). Danvy et al. have further studied the connections between the context-based and the higher-order approach by means of program-transformation techniques and they showed that these transformations can indeed be systematically obtained one from another (Danvy *et al.*, 2007).

This work reports on a method for constructing certified higher-order, one-pass transformations and is illustrated with three variants of the CPS transformation: the call-by-value and eta-expanded CPS, the call-by-value and eta-reduced CPS, and the call-by-value CPS with generalized beta-reduction. We show that such transformations, often designed from scratch and difficult to reason about, can be obtained mechanically as provably correct programs extracted from logical proofs using the Curry-Howard paradigm and the support of a proof assistant in a simple framework that can be adapted to other languages and other transformations. The fact that we extract higher-order programs is due to the use of logical relations in the proof, an approach that enables us to provide reduction-free proofs and to identify the resulting program as a fusion of the naive translation with an instance of normalization by evaluation in the target CPS language (Berger, 1993; Berger *et al.*, 2006; Dybjer & Filinski, 2000). It has been demonstrated before how programs extracted from normalization proofs constructed using the reducibility approach can be seen as instances of normalization by evaluation: Berger and Schwichtenberg extracted a NbE normalizer from their proof of strong normalization for the lambda calculus (Berger *et al.*, 2006), and similarly, Biernacka et al. extracted NbE evaluators from proofs of weak head normalization proofs for the lambda calculus (Biernacka *et al.*, 2005). Furthermore, it has been shown that a context-based variant of the reducibility approach (which hinges on explicit representation of evaluation contexts in the semantics) gives rise to NbE evaluators in continuation-passing style by extraction (Biernacka & Biernacki, 2009a; Biernacka & Biernacki, 2009b). In the light of these results, the present work illustrates how given a naive encoding and a reducibility-based proof of the normalization property for the CPS language yields the extracted normalizer that can be seen as a fusion of Plotkin's naive transformation and a normalization-by-evaluation function for the CPS language.

We have carried out the development in Coq and we used its extraction mechanism to generate certified programs realizing the transformations. We used Coq version 8.4 which offers extraction to OCaml, Haskell and Scheme (Team, 2014; Letouzey, 2002).

Related work. Several authors have formalized CPS transformations before. In Coq, Dargaye and Leroy have verified the correctness of a variant of Danvy and Nielsen's first-order transformation for a subset of the lambda calculus as part of a proof of correctness of a compiler for a small functional language (Dargaye & Leroy, 2007). They put emphasis on proving correctness and their approach is based on a big-step evaluation relation; as a byproduct they generate OCaml code implementing the corresponding transformation. Minamide and Okuma have formalized and verified several of the known one-pass CPS transformations in Isabelle/HOL (Minamide & Okuma, 2003), and Tian has carried out a similar development in Twelf (Tian, 2006). Their focus is on proving correctness and

they do not obtain transformations from the proofs. In contrast, our approach is based on small-step reduction, and we deliberately use the logical relations approach to proving normalization. As a result we obtain exactly Danvy and Filinski's higher-order one-pass transformation by extraction. Moreover, we show how well-defined modifications of the axioms defining the normalization in the CPS language lead to some other variants of the transformation.

1.1 An overview of the method of proof

The starting point for our development is the syntax of the source and the target languages, and an encoding from one to the other defined as a translation function. The translation need not be optimized, i.e., it may contain so-called administrative redices that could be contracted at compile time. For the target language, we specify a small-step operational semantics that reduces only administrative redices, and we then prove that for each term in the image of the encoding there exists a normal form with respect to the semantics. The proof is carried out constructively, and by the Curry-Howard isomorphism its computational content can be extracted as a functional program that implements an optimized translation without administrative reductions. We illustrate the method with three variants of call-by-value translation into CPS, and we identify the extracted programs as the known higher-order, one-pass transformations. We also discuss how the proof method affects the form of the resulting programs.

2 A higher-order one-pass CPS transformation

The source language for our development is the call-by-value lambda calculus with terms defined in the usual way:

$$t ::= x \mid \lambda x.t \mid t@t$$

where variables are drawn from a countably infinite set $\mathbf{V}$. We use an explicit application operator @ rather than juxtaposition in order to distinguish abstract-syntax constructors from transformation-time applications (denoted by $\overline{@}$) that will occur in the extracted programs. We assume the usual syntactic conventions about lambda terms without recalling them here.

Plotkin's call-by-value CPS translation introduces a continuation, i.e., a functional representation of "the rest of the computation" as a way to sequentialize computation and to name the intermediate results. All calls in a CPS-translated term are tail calls. The translation is as follows:

$$\begin{aligned}
[\![x]\!] &= \lambda k.k@x \\
[\![\lambda x.t]\!] &= \lambda k.k@(\lambda x.[\![t]\!]) \\
[\![t_0@t_1]\!] &= \lambda k.[\![t_0]\!](\lambda w_0.[\![t_1]\!](\lambda w_1.(w_0@w_1)@k))
\end{aligned}$$

The target language of the CPS transformation is a subset of lambda terms, and it can be characterized syntactically as follows:

$$\begin{aligned}
\text{(root terms)} \quad & r ::= \lambda k.e \\
\text{(expressions)} \quad & e ::= c@v \mid r@c \mid (v_0@v_1)@c \\
\text{(values)} \quad & v ::= \lambda x.r \mid x \mid w \\
\text{(continuations)} \quad & c ::= k \mid \lambda w.e
\end{aligned}$$

The result of CPS-transforming a lambda term is a root term, which expects a continuation. A continuation is either a continuation variable, or a function that expects a value. CPS values include (translated) source-term lambda abstractions and variables of two kinds: source-term variables (denoted x) and fresh variables introduced by the translation (denoted w).

The grammar of CPS terms allows us to identify and distinguish syntactically between source-term β-redices and administrative β-redices introduced by the translation. The former kind of redex is completely characterized as an application of a value to a value, because source-term lambda abstractions are translated to lambda abstractions in the CPS category of values, and any lambda abstraction in this category can only be obtained by translating a source-term lambda abstraction. The latter kind of redex can be either an application of a root term to a continuation $(\lambda k.e)@c$, or an application of a continuation to a value $(\lambda w.e)@v$. (We could introduce annotations on lambdas to distinguish the three kinds of lambda abstractions in the CPS grammar, but we prefer to keep things simple and identify a lambda abstraction by looking at its bound variable). Based on the syntactic distinction between residual and administrative redices we can write a reduction relation on CPS terms that only reduces administrative redices, and leaves the residual ones intact.

A one-pass CPS transformation reduces administrative redices away at translation time and produces compact CPS terms that can be defined using the following grammar:

$$\begin{aligned}
r &::= \lambda k.e \\
e &::= k@v \mid (v_0@v_1)@c \\
v &::= \lambda x.r \mid x \mid w \\
c &::= k \mid \lambda w.e
\end{aligned}$$

In contrast to full CPS grammar, the grammar of compact terms does not allow administrative redices as characterized above. It can be observed in the restricted grammar of expressions (applications of continuation variables to values are allowed since they do not form redices).

A one-pass variant of Plotkin's CPS transformation has been devised independently by Appel (Appel, 1992), Danvy and Filinski (Danvy & Filinski, 1990; Danvy & Filinski, 1992), and Wand (Wand, 1991). It is defined in the following way, using the two-level lambda calculus (Nielson & Nielson, 1992):

$$
\begin{array}{rcll}
T(t) & = & \lambda k.T_c(t,\overline{\lambda}u.k@u) & \text{where } k \text{ is fresh} \\
T_c(x,k) & = & k\overline{@}x & \\
T_c(\lambda x.t,k) & = & k\overline{@}(\lambda x.\lambda k'.T_c(t,\overline{\lambda}u.k'@u)) & \text{where } k' \text{ is fresh} \\
T_c(t_0@t_1,k) & = & T_c(t_0,\overline{\lambda}u_0.T_c(t_1,\overline{\lambda}u_1.(u_0@u_1)@(\lambda w.k\overline{@}w))) & \text{where } w \text{ is fresh}
\end{array}
$$

The translation into CPS is done by the function T and it uses an auxiliary function T_c called with an initial continuation. In the definition, residual lambda abstractions and applications are expressed with the constructs of the source language. An overlined lambda denotes a continuation that will be applied at translation time, and $k\overline{@}\ldots$ is such an application; it produces an administrative redex when a continuation is substituted for k and all such redices will be reduced away at translation time.

3 Proof of normalization in the CPS language

In this section we formalize the normalization process that leads to the elimination of administrative redices in the intermediate CPS terms. We characterize it as a sequence of one-step reductions and we prove that normalization always terminates using Tait's method based on reducibility predicates. Specifically, we prove the constructive *existence* of a CPS normal form, and we obtain the actual function computing it by extraction from the proof. Informally, we can think of a constructive proof of normalization as a function returning for each source term a CPS normal form together with the proof that it is indeed the correct normal form. The extraction procedure can then be seen as removing the logical parts and returning only the computational part of the proof.

The intensional structure of the extracted normalizer depends on the method of proof: it is possible to obtain different programs computing the same function from different normalization proofs. Here, we identify the computational content of the normalization proof with eta-expansion as exactly the known higher-order one-pass transformations T shown in Section 2. We then show that dropping the eta-expansion axiom leads to a different program that we show in Section 3.2. We discuss some of the relevant details of the proof and their connection to the extracted programs.

3.1 Axiomatization

We now present the small-step semantics for the CPS language. The normalization proof reported in this article matches the associated Coq formalization, while its presentation is meant to be human-readable, so it is more abstract than Coq code.

We give an axiomatization of the small-step reduction through an inference system presented in Figure 1. The first three axioms define the notion of reduction. The axioms $(\beta_r),(\beta_c)$ define a single computation step – an administrative β-reduction, and (η_c) defines η-expansion of continuation variables. The adoption of the η-rule allows for uniform treatment of all continuations as functions in the proof and it leads to eta-expanded continuation variables in the extracted program. As typical, we use the notation $e[k \mapsto c]$ to denote the capture-free substitution of c for the continuation variable k in e, and similarly

$$\frac{}{(\lambda k.e)@c \to_\beta e[k \mapsto c]}\,(\beta_r) \qquad \frac{}{(\lambda w.e)@v \to_\beta e[w \mapsto v]}\,(\beta_c) \qquad \frac{}{k \to_\eta \lambda w.k@w}\,(\eta_c)$$

$$\frac{e \to_\beta e'}{e \to_{\mathrm{exp}} e'}\,(e_1) \qquad \frac{v \to_{\mathrm{val}} v'}{k@v \to_{\mathrm{exp}} k@v'}\,(e_2) \qquad \frac{v_0 \to_{\mathrm{val}} v_0'}{(v_0@v_1)@c \to_{\mathrm{exp}} (v_0'@v_1)@c}\,(e_3)$$

$$\frac{v_1 \to_{\mathrm{val}} v_1'}{(v_0@v_1)@c \to_{\mathrm{exp}} (v_0@v_1')@c}\,(e_4) \qquad \frac{c \to_{\mathrm{cnt}} c'}{(v_0@v_1)@c \to_{\mathrm{exp}} (v_0@v_1)@c'}\,(e_5)$$

$$\frac{c \to_\eta c'}{c \to_{\mathrm{cnt}} c'}\,(c_1) \qquad \frac{e \to_{\mathrm{exp}} e'}{\lambda w.e \to_{\mathrm{cnt}} \lambda w.e'}\,(c_2)$$

$$\frac{r \to_{\mathrm{trm}} r'}{\lambda x.r \to_{\mathrm{val}} \lambda x.r'}\,(v) \qquad \frac{e \to_{\mathrm{exp}} e'}{\lambda k.e \to_{\mathrm{trm}} \lambda k.e'}\,(r)$$

Fig. 1. A small-step axiomatization of administrative normalization of CPS terms

for the substitution of values for term variables. The remaining axioms define the compatible closure of the notion of reduction with respect to expression, value, continuation and term constructors. In the Coq implementation, the corresponding relations are defined mutually inductively. The semantics is nondeterministic (the three rules $(e_3), (e_4), (e_5)$ can be applied in any order), but it is easily seen to be confluent.

Next, we define normalization predicates for each of the syntactic categories as the reflexive-transitive closure of the corresponding one-step relation, i.e., we write $\to_X^*$ for the reflexive-transitive closure of a relation $\to_X$. Furthermore, we use the following notation:

$$\begin{array}{llllll} r_0 \Downarrow_{\mathrm{trm}} r_1 & \text{iff} & r_0 \to_{\mathrm{trm}}^* r_1 & \text{and} & \mathrm{NF}(r_1) \\ e_0 \Downarrow_{\mathrm{exp}} e_1 & \text{iff} & e_0 \to_{\mathrm{exp}}^* e_1 & \text{and} & \mathrm{NF}(e_1) \\ v_0 \Downarrow_{\mathrm{val}} v_1 & \text{iff} & v_0 \to_{\mathrm{val}}^* v_1 & \text{and} & \mathrm{NF}(v_1) \\ c_0 \Downarrow_{\mathrm{cnt}} c_1 & \text{iff} & c_0 \to_{\mathrm{cnt}}^* c_1 & \text{and} & \mathrm{NF}(c_1) \end{array}$$

The predicate $\mathrm{NF}(\cdot)$ is true for terms, expressions, values and continuations in normal form with respect to the defined reduction relation.

We now define reducibility predicates à la Tait in order to prove the selective normalization of administrative redices. To this end, we treat each syntactic category as a type: values and expressions play the role of base types, and the categories of continuations and root terms behave like function types: when a continuation or a root term is applied to an argument, the resulting redex will be reduced (values and expressions do not generate redices this way). The definitions of the corresponding logical relations are thus the following:

$$\begin{aligned} \mathbf{R_v}(v) &\stackrel{\mathrm{df}}{=} \exists v'.\, v \Downarrow_{\mathrm{val}} v' \\ \mathbf{R_e}(e) &\stackrel{\mathrm{df}}{=} \exists e'.\, e \Downarrow_{\mathrm{exp}} e' \\ \mathbf{R_c}(c) &\stackrel{\mathrm{df}}{=} \forall v.\, \mathbf{R_v}(v) \to \mathbf{R_e}(c@v) \\ \mathbf{R_r}(r) &\stackrel{\mathrm{df}}{=} \forall c.\, \mathbf{R_c}(c) \to \mathbf{R_e}(r@c) \end{aligned}$$

We say that a continuation is reducible ($\mathbf{R_c}$) if, when it is applied to a reducible value, the resulting expression is reducible. Similarly, a term is reducible ($\mathbf{R_r}$) if, when it is applied

to a reducible continuation, the resulting expression is reducible. Values and expressions are reducible if they normalize.

The next step is to prove a theorem that asserts existence of CPS normal forms for all source terms translated into CPS terms by Plotkin's encoding $[\![\cdot]\!]$.

Theorem 1. *For each term $t \in \Lambda$, there exists a CPS term r (a term in CPS-normal form) such that $[\![t]\!] \Downarrow_{trm} r$ holds (i.e., $[\![t]\!]$ normalizes to r).*

Not surprisingly, the proof follows the idea and structure of the normalization proof for the simply typed lambda calculus using Tait's reducibility predicates (Martin-Löf, 1975; Girard *et al.*, 1989). It uses two key lemmas:

Lemma 1. *For each term $t \in \Lambda$, $\mathbf{R_r}([\![t]\!])$ holds.*

Lemma 2. *For each term r such that $\mathbf{R_r}(r)$, $r \Downarrow_{trm} r'$ holds for some r'.*

Lemma 1 states that the CPS encoding of a source term is reducible, and by Lemma 2 we then deduce that all terms have CPS normal forms (i.e., the corresponding compact CPS terms without administrative redices).

We have formalized the proof of Theorem 1 in the Coq proof assistant. We then applied the Coq extraction mechanism to obtain an OCaml program that implements the underlying normalization procedure. This extracted program coincides with the function T of Section 2 except for some Coq-specific artefacts arising in the extraction process.

The proof of Lemma 1 proceeds by induction on the structure of source terms. Whenever we need to apply the induction hypothesis, we have to specify the appropriate reducible continuation – for this the proof relies on Coq's automation mechanism and Prolog-like reasoning with generation of unification variables to be instantiated later. Moreover, we make sure not to inspect the structure of continuations c arising during the proof, but only use the fact that $\mathbf{R_c}(c)$ holds when appropriate. This fact has consequences for the extracted program that we obtain from the proof: the computational content of $\mathbf{R_c}(c)$ is a proper continuation, and if we do not inspect the structure of c then we can erase it from the extracted program as an unused argument. Specifically, the computational content of the predicate $\mathbf{R_c}$ has the following OCaml type:

```
type rc = val -> exp
```

which is the type of continuations expecting CPS values and returning CPS expressions (in normal form). The computational contents of the remaining predicates have the following OCaml types:

```
type rv = val
type re = exp
type r = (val -> exp) -> exp
```

These types correspond to those extracted by the Coq extraction procedure but are simplified here by removing unused parts of the extracted types. The types `val` and `exp` are extracted types of values and expressions of the CPS language, respectively.

3.2 Eta-reduced CPS transformation

Let us now consider the role of the (η_c)-axiom in the axiomatization of Figure 1. As mentioned before, adopting this axiom leads to eta-expanded continuation variables in CPS, and consequently to exactly the one-pass transformer T shown in the previous section. Alternatively, we can discard the (η_c)-axiom in order to obtain eta-reduced continuations in CPS normal forms. At the level of proof, this variant requires inspection of the structure of continuation arguments and it leads to the extraction of the following normalization function:

$$\begin{array}{rcll}
T^\eta(t) & = & \lambda k.T_k^\eta(t,k) & \text{where } k \text{ is fresh} \\
T_c^\eta(x,c) & = & c\overline{@}x & \\
T_c^\eta(\lambda x.t,c) & = & c\overline{@}(\lambda x.\lambda k'.T_k^\eta(t,k')) & \text{where } k' \text{ is fresh} \\
T_c^\eta(t_0@t_1,c) & = & T_c^\eta(t_0,\overline{\lambda}u_0.T_c^\eta(t_1,\overline{\lambda}u_1.(u_0@u_1)@(\lambda w.c\overline{@}w))) & \text{where } w \text{ is fresh} \\
T_k^\eta(x,k) & = & k@x & \\
T_k^\eta(\lambda x.t,k) & = & k@(\lambda x.\lambda k'.T_k^\eta(t,k')) & \text{where } k' \text{ is fresh} \\
T_k^\eta(t_0@t_1,k) & = & T_c^\eta(t_0,\overline{\lambda}u_0.T_c^\eta(t_1,\overline{\lambda}u_1.(u_0@u_1)@k)) &
\end{array}$$

This normalizer coincides with Danvy and Filinski's tail-conscious CPS transformation (Danvy & Filinski, 1992).

4 Context-sensitive administrative reductions

It is possible to obtain even more compact CPS terms by performing more administrative reductions. For example, Danvy and Nielsen (Danvy & Nielsen, 2005) consider additional administrative redices occurring in the translation of source β-redices. The source term $(\lambda x.t)@v$ is translated to a call-by-value CPS term of the form $((\lambda x.r)@v')@c$, and the dynamic redex $(\lambda x.r)@v'$ here blocks further administrative redices involving r and c. We can remedy the situation by introducing an extra reduction rule in the CPS language that moves the continuation c into the body of the abstraction, thus bypassing the dynamic redex and triggering administrative reductions. Danvy and Nielsen also point out that this reduction step corresponds to Sabry and Felleisen's source-level reduction of the form

$$E[(\lambda x.t)@t'] \to (\lambda x.E[t])@t',$$

where E is a nonempty reduction context and x is not free in E(Sabry & Felleisen, 1993).

This new reduction can be characterized by the following axiom that extends the rules of Figure 1:

$$\frac{}{((\lambda x.r)@v)@c \to_\beta (\lambda x.r@c)@v}\,(\beta_{lift})$$

In order to account for this reduction, we need to extend the grammar of expressions and values with new constructs so that now it reads:

$$e ::= \ldots \mid v_0@v_1$$

$$v ::= \ldots \mid \lambda x.e$$

Furthermore, an additional rule for compatibility is needed:

$$\frac{e \to_{\text{exp}} e'}{\lambda x.e \to_{\text{val}} \lambda x.e'}\,(\nu_2)$$

The definitions of logical relations also have to be adjusted in the treatment of values. If a source lambda abstraction is in the operator position, it is no longer enough that it terminates – we need to ensure that the reductions triggered by the application of (β_{lift}) will terminate. Therefore we need to define the reducibility property for values in such a way as to express the following intuitive property:

$$\mathbf{R_v}(v) \stackrel{\text{df}}{=} \exists v'.\, v \Downarrow_{\text{val}} v' \wedge \forall v_0 c.\, (\exists v_0'.\, v_0 \Downarrow_{\text{val}} v_0') \to \mathbf{R_c}(c) \to \mathbf{R_e}((v@v_0)@c)$$

Unfortunately, this definition is not well-founded since the predicate $\mathbf{R_c}$ refers back to $\mathbf{R_v}$ (see the previous definition).

What we can do instead is to use an indexed logical relation that takes an additional parameter tracking the depth of the lambda abstraction in an application. We can then define the property "reducibility at level n": a value reducible at level n guarantees termination when applied to a terminating value and a continuation that only expects reducible values of levels smaller than n.

The complete definition of logical relations is now the following:

$$\mathbf{R_v}(0,v) \stackrel{\text{df}}{=} \exists v'.\, v \Downarrow_{\text{val}} v'$$

$$\mathbf{R_v}(n+1,v) \stackrel{\text{df}}{=} \forall v' c.\, \mathbf{R_v}(0,v') \to \mathbf{R_c}(n,c) \to \mathbf{R_e}((v@v')@c)$$

$$\mathbf{R_e}(e) \stackrel{\text{df}}{=} \exists e'.\, e \Downarrow_{\text{exp}} e'$$

$$\mathbf{R_c}(n,c) \stackrel{\text{df}}{=} \forall v.\, \mathbf{R_v}(n,v) \to \mathbf{R_e}(c@v)$$

$$\mathbf{R_r}(n,r) \stackrel{\text{df}}{=} \forall c.\, \mathbf{R_c}(n,c) \to \mathbf{R_e}(r@c)$$

We have redefined the $\mathbf{R_v}$ predicate and taken into account the level parameter in the definitions of $\mathbf{R_r}$ and $\mathbf{R_c}$. The definition of $\mathbf{R_e}$ does not change.

The statement of the normalization theorem remains the same and the proof proceeds along similar lines as before. We need to change the formulation of auxiliary lemmas as follows:

Lemma 3. *For each term $t \in \Lambda$ and for all $n \in \mathcal{N}$, $\mathbf{R_r}(n, [\![t]\!])$ holds.*

Lemma 4. *For each term r such that $\mathbf{R_r}(0,r)$ holds, $r \Downarrow_{trm} r'$ holds for some r' (CPS normal form).*

The proof of Lemma 4 does not change and the proof of Lemma 3 changes only in that it requires that we keep track of depth in applications: intuitively, when we talk about an operand we increase the index, and when we talk about the argument we set the index to 0.

The program extracted from the normalization theorem in this case reads as follows:

$$
\begin{array}{rcll}
T_{lift}(t) & = & \lambda k.T_c^0(t,\overline{\lambda}u.k@u) & \text{where } k \text{ is fresh} \\
T_c^n(x,k) & = & k\overline{@}(U^n(x)) & \\
T_c^0(\lambda x.t,k) & = & k\overline{@}(\lambda x.\lambda k'.T_c^0(t,\overline{\lambda}u.k'@u)) & \text{where } k' \text{ is fresh} \\
T_c^{n+1}(\lambda x.t,k) & = & k\overline{@}(\overline{\lambda}t'.\overline{\lambda}k'.(\lambda x.T_c^n(t,k'))@t') & \\
T_c^n(t_0@t_1,k) & = & T_c^{n+1}(t_0,\overline{\lambda}u_0.T_c^0(t_1,\overline{\lambda}u_1.u_0\overline{@}u_1\overline{@}k)) & \\
 & & & \\
U^0(v) & = & v & \\
U^{n+1}(v) & = & \overline{\lambda}t.\overline{\lambda}k.(v@t)@(\lambda w.k\overline{@}(U^n(w))) & \text{where } w \text{ is fresh}
\end{array}
$$

This program coincides with the normalizer of Danvy and Nielsen (Danvy & Nielsen, 2005).

A similar optimization can be done for the call-by-name CPS translation, where the new reduction rule reads as follows:

$$\frac{}{((\lambda x.r')@r)@c \to_\beta (\lambda x.r'@c)@r}\,(\beta^n_{lift})$$

We have obtained a call-by-name variant of the optimized translation along the same lines as for call by value.

5 Formalization in Coq

In this section we present the main features of the Coq formalization accompanying this article. The code is available at `http://bitbucket.org/mabi/onepass` and it is compatible with the Coq version 8.4. The repository contains Coq source files as well as files obtained by the Coq extraction mechanism from the proofs. These extracted files have been further annotated with comments in order to clarify the connection between the proofs and the extracted code.

5.1 Representation and structure

The formalization uses a deep embedding of both the source and the target CPS language. In consequence, we have to deal with variable bindings explicitly. We do this by using the locally nameless representation of terms – where bound variables are represented by de Bruijn indices and free variables are represented by names – and by applying the technique known as cofinite quantification implemented in Charguéraud's LN library (Aydemir *et al.*, 2008; Charguéraud, 2011). In particular, the library provides an interface for fresh variable generation that we use extensively: whenever we talk about the body of an abstraction as a proper term, we need to substitute a fresh name for the de Bruijn index of the bound variable. We then need to prove a number of technical lemmas that are needed for the handling of the representation of terms. Fortunately, the required lemmas have been formulated by Charguéraud in his library for several typical languages; their proofs can be adapted to our setting in a straightforward way, therefore the formalization overhead is not big.

The naive translation from source to CPS terms is implemented as a simple recursive function. To formalize the normalization problem in the CPS language we need to define the axioms of Figure 1 and the logical relations needed for the proof. The axioms are represented as mutually inductive definitions, and the logical relations are transliterated from their mathematical definition. They are, however, defined using subset types rather than existential formulas in order to allow for the extraction of witnesses from proofs.

The proofs of the auxiliary lemmas are done by induction, where the induction principles needed for each proof are Coq-generated for the appropriate inductive definition. Coq's tactic language supports automation in the process of constructing proofs; we have exploited this feature in order to increase modularity and facilitate adaptations of the formalization to different axiomatizations (e.g., big-step) of the same strategy, or even for different strategies (e.g., call-by-name).

5.2 Extracted programs

The extraction mechanism of Coq produces code in OCaml, Haskell or Scheme. The programs obtained by extraction are essentially the one-pass transformers shown in previous sections, except that they are cluttered with logical artefacts that could be eliminated. There are two main sources of clutter. First, the proofs in Coq use dependently typed objects whose computational content is not well typed in a non-dependent language, therefore the extracted code contains occurrences of unsafe coercions (in statically typed OCaml and Haskell). This can be observed in particular in the normalization proof for generalized beta reduction. The predicate $\mathbf{R_v}$ in this case is not definable inductively in Coq and is defined instead with a recursive function on natural numbers. Consequently, its computational content is a dependent type modeled by a special type in OCaml and the extraction process inserts coercions (`Obj.magic : 'a -> 'b`) when needed.

Second, the structure of the extracted code depends both on the method of proof and on the extraction algorithm used in Coq. The code can be further optimized by inlining and dead code elimination. Some of these optimizations can be prescribed already at the level of Coq formalization using special commands. Moreover, in our development the actual implementation of the procedure generating fresh variables (assumed as an axiom in the formalization) has to be provided in the extracted code. In OCaml, we can use a simple *gensym* function to achieve that. As a result, we can further optimize the program by removing the continuation argument from the main function because it is only used in the proof to control freshness of variables and not to construct CPS normal forms.

The fact that the extracted programs are higher-order and use continuations arises from the use of the logical relation $\mathbf{R_c}$ whose computational content is a continuation (a higher-order function), as shown in Section 3.1. The programs operate in one pass, i.e., they do not rely on constructing intermediate CPS terms – the images of the encoding. It is due to the fact that the proofs do not use the intermediate CPS terms for construction of the final result but only manipulate the reducibility predicates instead.

6 Conclusion

We have shown how to obtain higher-order, one-pass transformations from the lambda calculus to compact CPS terms by extracting them from proofs of the appropriate normalization property in the CPS language. The normalization proofs have been constructed by adapting the reducibility method due to Tait and defining suitable logical relations. In our case, we have identified the extracted programs as already known transformations, thus we have provided formal proofs of their correctness. The presented approach can also be used to obtain new, provably correct, one-pass transformations given a non-optimizing translation and a specification of normalizability in the target language. This method of writing these particular higher-order programs may be easier than having to write a function from scratch, since the ingredients needed in our approach are usually known for a given language, and it is a matter of putting together the pieces of the puzzle to mechanically obtain the code, possibly using a proof assistant to help automate the development. We also note that our programs are in each case instances of the naive translation fused with a normalization-by-evaluation function for the CPS language.

References

Appel, Andrew W. (1992). *Compiling with Continuations*. New York: Cambridge University Press.

Aydemir, Brian E., Charguéraud, Arthur, Pierce, Benjamin C., Pollack, Randy, & Weirich, Stephanie. (2008). Engineering formal metatheory. *Pages 3–15 of:* Necula, George C., & Wadler, Philip (eds), *Proceeding of the 35th ACM SIGLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM.

Berger, Ulrich. (1993). Program extraction from normalization proofs. *Pages 91–106 of:* Bezem, Marc, & Groote, Jan Friso (eds), *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, no. 664. Utrecht, The Netherlands: Springer-Verlag.

Berger, Ulrich, Berghofer, Stefan, Letouzey, Pierre, & Schwichtenberg, Helmut. (2006). Program extraction from normalization proofs. *Studia logica*, **82**(1), 25–49.

Biernacka, Małgorzata, & Biernacki, Dariusz. 2009a (Apr.). A context-based approach to proving termination of evaluation. *Proceedings of the 25th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXV)*.

Biernacka, Małgorzata, & Biernacki, Dariusz. (2009b). Context-based proofs of termination for typed delimited-control operators. López-Fraguas, Francisco J. (ed), *Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. Coimbra, Portugal: ACM Press.

Biernacka, Małgorzata, Danvy, Olivier, & Støvring, Kristian. (2005). Program extraction from proofs of weak head normalization. *Pages 169–189 of:* Escardó, Martin, Jung, Achim, & Mislove, Michael (eds), *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*. Electronic Notes in Theoretical Computer Science, vol. 155. Birmingham, UK: Elsevier Science Publishers.

Charguéraud, Arthur. (2011). The locally nameless representation. *Journal of Automated Reasoning*, 1–46. 10.1007/s10817-011-9225-2.

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of:* Wand, Mitchell (ed), *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. Nice, France: ACM Press.

Danvy, Olivier, & Filinski, Andrzej. (1992). Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, **2**(4), 361–391.

Danvy, Olivier, & Nielsen, Lasse R. (2003). A first-order one-pass CPS transformation. *Theoretical Computer Science*, **308**(1-3), 239–257. A preliminary version was presented at the Fifth International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002).

Danvy, Olivier, & Nielsen, Lasse R. (2005). CPS transformation of beta-redexes. *Information Processing Letters*, **94**(5), 217–224. Extended version available as the research report BRICS RS-04-39. A preliminary version was presented at the 2001 ACM SIGPLAN Workshop on Continuations (CW 2001).

Danvy, Olivier, Millikin, Kevin, & Nielsen, Lasse R. (2007). On one-pass CPS transformations. *Journal of Functional Programming*, **17**(6), 793–812.

Dargaye, Zaynah, & Leroy, Xavier. (2007). Mechanized verification of CPS transformations. *Pages 211–225 of: Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*. Lecture Notes in Artificial Intelligence, vol. 4790. Springer.

Dybjer, Peter, & Filinski, Andrzej. (2000). Normalization and partial evaluation. *Pages 137–192 of:* Barthe, Gilles, Dybjer, Peter, Pinto, Luís, & Saraiva, João (eds), *Applied Semantics – Advanced Lectures*. Lecture Notes in Computer Science, no. 2395. Caminha, Portugal: Springer-Verlag.

Girard, Jean-Yves, Lafont, Yves, & Taylor, Paul. (1989). *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University Press.

Letouzey, Pierre. (2002). A new extraction for Coq. Geuvers, Herman, & Wiedijk, Freek (eds), *Types for Proofs and Programs, International Workshop TYPES'02*. Lecture Notes in Computer Science, no. 2646. Berg en Dal, The Netherlands: Springer-Verlag.

Martin-Löf, Per. (1975). About models for intuitionistic type theories and the notion of definitional equality. *Pages 81–109 of: Proceedings of the Third Scandinavian Logic Symposium (1972)*. Studies in Logic and the Foundation of Mathematics, vol. 82. North-Holland.

Minamide, Yasuhiko, & Okuma, Koji. 2003 (Sept.). Verifying CPS transformations in Isabelle/HOL. Momigliano, Alberto, & Miculan, Marino (eds), *Proceedings of the Second ACM SIGPLAN Workshop on MEchanized Reasoning about Languages with varIable BiNding*. http://merlin.dimi.uniud.it/merlin03/.

Nielson, Flemming, & Nielson, Hanne Riis. (1992). *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science, vol. 34. Cambridge University Press.

Plotkin, Gordon D. (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**, 125–159.

Sabry, Amr, & Felleisen, Matthias. (1993). Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, **6**(3/4), 289–360. A preliminary version was presented at the 1992 ACM Conference on Lisp and Functional Programming (LFP 1992).

Team, The Coq Development. (2014). *The Coq Proof Assistant Reference Manual, Version 8.4pl4*. INRIA.

Tian, Ye Henry. (2006). Mechanically verifying correctness of CPS compilation. *Pages 41–51 of: CATS'06: Proceedings of the 12th computing: The Australasian Theory Symposium*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc.

Wand, Mitchell. (1991). Correctness of procedure representations in higher-order assembly language. *Pages 294–311 of:* Brookes, Stephen, Main, Michael, Melton, Austin, Mislove, Michael, & Schmidt, David (eds), *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*. Lecture Notes in Computer Science, no. 598. Pittsburgh, Pennsylvania: Springer-Verlag.