# Generating certified higher-order one-pass transformations

Małgorzata Biernacka
Institute of Computer Science
University of Wrocław
mabi@cs.uni.wroc.pl

**Abstract**

We show a method of mechanical derivation of provably correct higher-order and one-pass transformations based on the Curry-Howard paradigm. We illustrate the method with two known higher-order, one-pass transformations: one from the (call-by-value) lambda calculus to monadic normal forms, and the other from the (call-by-value) lambda calculus to compact CPS terms. The verification in each case is achieved by proving a theorem from which an automatic method of program extraction yields exactly the desired transformation. Secondly, the method of proof in each case uses a particular form of logical relations whose computational content is a continuation (or a functional accumulator) used in the transformations. The proposed method is a recipe for obtaining certified one-pass transformations automatically, given a naive, non-optimizing encoding from a source to a target language, and a method of proof of normalization à la Tait in the target language. Depending on the method of proof, one could obtain other one-pass transformations for the same language; our focus is on higher-order ones, which are also identified to be instances of normalization by evaluation. The development in each case has been carried out in the Coq system and from the proofs we obtained Ocaml/Haskell programs implementing the two transformations.

## 1 Introduction

Compilers are a typical tool where program transformations are made from one language to another. Depending on the choice of the implementor, such transformations can be simple, well-understood translations generating code that is subsequently subject to various optimizations (compilation operates in two passes), or they can be defined as translations mapping source code directly to optimized target code, thus operating in one pass.

In this article we focus on two of such translations of widespread use: a higher-order, one-pass transformation from lambda terms to monadic normal forms, and to compact CPS terms. While the naive translations from the source

to the corresponding target language in each case is well-known and understood, the task of devising corresponding one-pass transformations has proven to be more complex even though its extensional behaviour is characterized in a simple manner, namely as the non-optimizing (naive) translation composed with subsequent normalization in the target code, i.e., reducing "administrative redexes" introduced by the naive translation. The normalization phase should only reduce the administrative redexes and not the ones present in the translated source term. This characterization has led to two-level, higher-order executable specifications of one-pass transformations, building on Plotkin's naive translation [23]. For example the first higher-order, one-pass transformation to compact CPS terms have been proposed independently by Appel [1], Danvy and Filinski [9, 10] and Wand [28]. Another approach is to define first-order, context-based transformations [11, 24]. Recently, Danvy et al. have studied the connections between the context-based and the higher-order approach by means of program transformation techniques [12]. In this paper we consider Danvy and Filinski's transformation [9, 10].

The monadic metalanguage is inspired by Moggi's computational lambda calculus and is also a popular choice in compiler implementations [3, 17, 22, 25]. It offers similar advantages to CPS, i.e., it sequentializes computation (term application) and binds its results to names that can be used in further computations or returned. A higher-order, one-pass transformation to monadic normal forms, including efficient shortcut boolean evaluation has been devised by Danvy [8]. In this article we only consider the lambda calculus without booleans, but we anticipate that the method presented extends seamlessly (yet tediously) to the full language and its transformation. We are also convinced the method works for A-normal forms [15, 17].

The goal of this work is not so much the verification of correctness of the mentioned transformations (in a novel way), as it is to show how such transformations, often painstakingly devised and difficult to reason about, can be obtained mechanically as provably correct programs extracted from logical proofs using the Curry-Howard paradigm (and support of a proof assistant). Our approach combines results from proof theory (computational content of normalization proofs) and programming language theory. In particular, the fact of extracting higher-order programs is due to the use of logical relations in the proof, enabling us to provide reduction-free proofs and thus, the resulting program can be seen to be composed of an instance of normalization by evaluation in the target language [4, 6, 14]. As evidence, we have carried out the development in the Coq system of interactive proof development and we used its extraction mechanism to generate programs realizing the transformations[1] [19, 26].r

Recently, Dargaye and Leroy have mechanically verified in Coq 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 [13]. They put emphasis on proving correctness and their

[1] We used Coq version 8.1pl3 of December 2007, which offers extraction to Ocaml, Haskell and Scheme.

approach is based on big-step evaluation relation, and as a byproduct they generates Ocaml code implementing the corresponding transformation. In contrast, our approach is small-step and uses logical relations to obtain exactly Danvy and Filinski's higher-order program by extraction. Minamide and Okuma have formalized and verified several of the known one-pass CPS transformations in Isabelle/HOL [21], and Tian have carried out a similar development in Twelf [27]. Their focus is on proving correctness and they do not obtain transformations from the proofs.

### 1.1 An overview of the method of proof

The starting point for our development in both cases is the syntax of the source language (here, in both cases the lambda calculus), a specification of the target language and an encoding from one to the other defined as a translation function. The result of this translation need not be optimal, i.e., it may contain some "administrative redexes" that could be eliminated at compile-time. The target language in each case will be given a small-step operational semantics (reduction semantics) and the result we prove is that for each term in the target language that is in the image of the encoding there exists a normal form with respect to the given reduction semantics. Both proofs will be carried out constructively, and by the Curry-Howard isomorphism, their computational content can be extracted as a functional program. 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.

#### 1.1.1 Plan of the article.

In Section 2 we detail the development for the monadic target language. We first express the problem and the proof informally, and then discuss the Coq implementation and the extraction mechanism. We compare the Coq extraction mechanism with the modified realizability interpretation, since the latter is more flexible and can be used in the present case to obtain exactly the transformation we start from. In Section 3, we outline a similar development for the CPS target language.

## 2 A monadic transformation

In this section we first recall Danvy's higher-order one-pass transformation from the lambda calculus to monadic normal form [8]. Next, we state its correctness and give an informal proof. The problem and its proof is expressible essentially in first-order logic, except for induction on terms that in the first-order setting has to be done at the meta-level. We have done the formal proof in the Coq system, which is based on higher-order logic with inductive definitions, where most of the axioms assumed below can be implemented using inductive definitions. We include a brief discussion of the Coq formalization at the end of the

section. A crucial point in the development is the extraction of a program from the proof – the extraction from (partly) formalized proof in first-order logic can be done using modified realizability interpretation with an optimization due to Berger which eliminates unused input variables. This kind of extraction yields exactly the program that we start with. In Coq, the extraction mechanism is less flexible and such optimization is not implemented (but it can be done by hand on the extracted program).

The source language $\Lambda$ 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 assume the usual syntactic conventions about lambda terms without recalling them here, since we do not manipulate lambda terms directly.

The target language $\Lambda^{\mathrm{m}}$ is that of monadic terms defined by the following grammar, where variables are again elements of the set $\mathbf{V}$:

$$\begin{aligned} \text{(expressions)} \quad & e ::= \mathtt{ret}\, v \mid v_0\, v_1 \mid \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 \\ \text{(values)} \quad & v ::= x \mid \lambda x.e \end{aligned}$$

The let-construct $\mathtt{let}\ x = e_0\ \mathtt{in}\ e_1$ binds the variable $x$ in $e_1$ (it corresponds to the monadic operator `bind`) and the construct $\mathtt{ret}\, v$ injects a value into a computation (it corresponds to the monadic operator `return`). For simplicity of presentation, we will later assume that let-bound variable names are fresh in that they do not occur in source terms being translated. Alternatively, we could use a mixed approach and use, e.g., de Bruijn indices to represent let-bound variables and names for object-level variables.

For reasons that will become apparent later on, we single out the syntactic category of let-contexts and we restate the above grammar in the following way:

$$\begin{aligned} \text{(expressions)} \quad & e ::= \mathtt{ret}\, v \mid v_0\, v_1 \mid c[e] \\ \text{(let-contexts)} \quad & c ::= \mathtt{let}\ x = [\,]\ \mathtt{in}\ e \\ \text{(values)} \quad & v ::= x \mid \lambda x.e \end{aligned}$$

An expression of the form $c[e]$ is then understood as meta-notation for the result of plugging the context $c$ with expression $e$, therefore syntactically equal to $\mathtt{let}\ x = e\ \mathtt{in}\ e_0$, where $c := \mathtt{let}\ x = [\,]\ \mathtt{in}\ e_0$. We use the first notation whenever we do not need to know the structure of $c$.

The terms produced by the one-pass transformation are normal forms in $\Lambda^{\mathrm{m}}$, i.e., terms characterized by the following grammar:

$$\begin{aligned} \text{(normal expressions)} \quad & e ::= \mathtt{ret}\, v \mid v_0\, v_1 \mid c[v_0\, v_1] \\ \text{(normal contexts)} \quad & c ::= \mathtt{let}\ x = [\,]\ \mathtt{in}\ e \\ \text{(normal values)} \quad & v ::= x \mid \lambda x.e \end{aligned}$$

We can see that normal expressions are either encapsulated normal values, an application of values in tail position, or a let-expression, in which the

computation bound to a variable is an application of values, i.e., a "simple" computation.

Let us now focus on the call-by-value encoding from the lambda calculus to the monadic language (the call-by-name case can be handled in a similar way). The translation is defined recursively over the structure of source terms and consists in sequentializing computation and naming the intermediate results in the following way:

$$\mathcal{E}(\,\mathtt{x}) = \mathtt{ret}\,\mathtt{x}$$

$$\mathcal{E}(\,\lambda\mathtt{x}.\mathtt{t}) = \mathtt{ret}\,(\lambda\mathtt{x}.\mathcal{E}(\,\mathtt{t}))$$

$$\mathcal{E}(\,\mathtt{t}_0\,\mathtt{t}_1) = \mathtt{let}\ w_0 = \mathcal{E}(\,\mathtt{t}_0)\ \mathtt{in}\ \mathtt{let}\ w_1 = \mathcal{E}(\,\mathtt{t}_1)\ \mathtt{in}\ w_0\,w_1$$

In the translation we use two distinguished variables: $w_0$ and $w_1$, for which we assume that they do not occur in source terms – this property will be needed in the proof in order to avoid $\alpha$-renaming when performing monadic reductions (such renaming greatly obscures the proof). Either approach we use to handling generation of fresh variables, the decision is orthogonal to the proof method (and to the extraction of program), therefore we do not dwell on this issue here. In the Coq development, we also follow the freshness assumption.

The result of the encoding $\mathcal{E}$ contains administrative redexes that can be eliminated at compile-time. These redexes are the monadic redexes of the language $\Lambda^{\mathrm{m}}$:

$$\mathtt{let}\ \mathtt{x} = \mathtt{ret}\ v\ \mathtt{in}\ e \longrightarrow e[\mathtt{x} \mapsto v]$$

$$\begin{array}{lll} \mathtt{let}\ \mathtt{x} = \ \mathtt{let}\ \mathtt{y} = e_0 & & \mathtt{let}\ \ \mathtt{y} = e_0 \\ \qquad\quad\ \mathtt{in}\ e_1 & \longrightarrow & \mathtt{in}\ \ \mathtt{let}\ \mathtt{x} = e_1 \\ \mathtt{in}\ e_2 & & \qquad\quad\ \mathtt{in}\ e_2 \end{array}$$

if $\mathtt{y} \notin \mathbf{FV}\,(e_2)$

The first reduction is essentially $\beta$-reduction, replacing an already computed value in the body of a let-expression. The substitution operation $e[\mathtt{x} \mapsto v]$ denotes the result of (capture-avoiding) replacement of all free occurrences of variable $\mathtt{x}$ in $e$ with $v$. In our case, we do not reduce arbitrary monadic terms, but only those in the image of the encoding, hence our assumption about global freshness of the distinguished variables is enough to ensure that substitution is a capture-free operation. The second reduction is the so-called commuting conversion that performs the "unnesting" of let-expressions.

These reductions can be performed in reduction contexts, where reduction contexts are standard and can be understood for example as terms (expressions) with a hole:

$$\mathtt{E} ::= \cdot \mid \mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ \mathtt{E}$$

Thus, a second transformation eliminating administrative redexes in a monadic term can be performed after the encoding, yielding terms in monadic normal form. Alternatively, one can directly write a one-pass transformation, e.g., as presented in Danvy [8], mapping source terms directly to monadic normal

forms. This transformation is higher-order and uses continuations[2] to direct the reordering of nested let constructors. In order to distinguish object level and meta-level application, we annotate the latter as $\kappa@\nu$ rather than $\kappa\,\nu$ and meta-level lambda abstractions are denoted $\overline{\lambda}$ rather than $\lambda$:

$$\begin{aligned}
&\mathcal{E}'\,(x) = (\mathtt{ret}\,x)\\
&\mathcal{E}'\,(\lambda x.t) = (\mathtt{ret}\,\lambda x.\mathcal{E}'(\,t))\\
&\mathcal{E}'\,(t_0\,t_1) = \mathcal{E}'_c\,(t_0\,,\overline{\lambda}\nu_0.\mathcal{E}'_c\,(t_1\,,\overline{\lambda}\nu_1.\nu_0\,\nu_1))\\
&\mathcal{E}'_c\,(x\,,\kappa) = \kappa@(\mathtt{ret}\,x)\\
&\mathcal{E}'_c\,(\lambda x.t\,,\kappa) = \kappa@(\mathtt{ret}\,(\lambda x.\mathcal{E}'(\,t)))\\
&\mathcal{E}'_c\,(t_0\,t_1\,,\kappa) =\\
&\quad \mathcal{E}'_c\,(t_0\,,\overline{\lambda}\nu_0.\mathcal{E}'_c\,(t_1\,,\overline{\lambda}\nu_1.\mathtt{let}\;w = \nu_0\,\nu_1\;\mathtt{in}\;\kappa@w))
\end{aligned}$$

In the remainder of this section, we will verify the correctness of the function $\mathcal{E}'$ with respect to monadic reductions, i.e., we will prove that the result of applying $\mathcal{E}'$ on a term $t$ is indeed the normal form of its image through the encoding. However, rather than proving this property directly, we only prove the (constructive) *existence* of such a normal form, and we obtain the actual function computing it by extraction from the proof. This extracted function coincides with the function $\mathcal{E}'$. Therefore, we do not only verify that the function is correct in the given sense, but we also show how to obtain a one-pass transformation automatically if we know how to write the encoding and if we construct the proof of the theorem. The form of the function that we extract depends in an essential way on the method of proof and we pick the one that leads to the one-pass, higher-order function that we focus on, i.e., using logical relations describing reducibility properties of language constructs.

## 2.1 The proof of existence of monadic normal forms

First we need to set up the stage for our proof, i.e., we introduce all the ingredients needed to express the normalization property for monadic terms, which we will then prove. To this end, we formalize the small-step reduction semantics for the monadic language as logical properties.

Let $\mathbf{N}(e_0, e_1)$ be a logical predicate expressing the property that $e_0$ normalizes to $e_1$ in $\Lambda^m$. Eventually, we would like to prove that $\mathbf{N}(\mathcal{E}(\,t), \mathcal{E}'(\,t))$ holds for all source terms $t$, where $\mathcal{E}$ denotes the naive translation of lambda to monadic terms, and $\mathcal{E}'$ denotes the optimized translation (operating in one-pass). We give a logical characterization of the predicate $\mathbf{N}$ through axioms essentially defining the normalization relation in $\Lambda^m$. What is important to note is that these axioms do not carry any computational meaning – they are

[2]The function $\mathcal{E}'_c$ is not in continuation-passing style because the call to continuation $\kappa$ in the last clause is not a tail call. Therefore some would prefer to call $\kappa$ a functional accumulator rather than a continuation.

**Axiom 1.** $\forall \mathtt{x}.\, \mathbf{A}(\mathtt{x}, \mathtt{x})$

**Axiom 2.** $\forall \mathtt{x} e_0 e_1.\, \mathbf{N}(e_0, e_1) \to \mathbf{A}(\lambda \mathtt{x}.e_0, \lambda \mathtt{x}.e_1)$

**Axiom 3.** $\forall v_0 v_1.\, \mathbf{A}(v_0, v_1) \to \mathbf{N}(\mathtt{ret}\, v_0, \mathtt{ret}\, v_1)$

**Axiom 4.** $\forall v_0 w_0 v_1 w_1.\, \mathbf{A}(v_0, v_1) \to \mathbf{A}(w_0, w_1) \to \mathbf{N}(v_0\, w_0, v_1\, w_1)$

**Axiom 5.** $\forall v_0 w_0 v_1 w_1 \mathtt{c}_0 \mathtt{c}_1.\, \mathbf{A}(v_0, v_1) \to \mathbf{A}(w_0, w_1) \to \mathbf{C}(\mathtt{c}_0, \mathtt{c}_1) \to \mathbf{N}(\mathtt{c}_0[v_0\, w_0], \mathtt{c}_1[v_1\, w_1])$

**Axiom 6.** $\forall \mathtt{x} e_0 e_1.\, \mathbf{N}(e_0, e_1) \to \mathbf{C}(\mathtt{let}\ \mathtt{x} = [\,]\ \mathtt{in}\ e_0, \mathtt{let}\ \mathtt{x} = [\,]\ \mathtt{in}\ e_1)$

**Axiom 7.** $\forall e_0 e_1 e_2.\, \mathbf{Rd}(e_0, e_1) \to \mathbf{N}(e_1, e_2) \to \mathbf{N}(e_0, e_2)$

**Axiom 8.** $\forall \mathtt{x} v e.\, \mathbf{Rd}(\mathtt{let}\ \mathtt{x} = \mathtt{ret}\, v\ \mathtt{in}\ e, e[\mathtt{x} \mapsto v])$

**Axiom 9.** $\forall \mathtt{x}\mathtt{y} e_0 e_1 e_2.\, \mathbf{Rd}(\mathtt{let}\ \mathtt{x} = (\mathtt{let}\ \mathtt{y} = e_0\ \mathtt{in}\ e_1)\ \mathtt{in}\ e_2, \mathtt{let}\ \mathtt{y} = e_0\ \mathtt{in}\ \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2)$ *if* $\mathtt{y} \notin \mathbf{FV}(e_2)$

**Axiom 10.** $\forall \mathtt{x} e e_0 e_1.\, \mathbf{Rd}(e_0, e_1) \to \mathbf{Rd}(\mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ e_0, \mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ e_1)$

Figure 1: Axiomatization of normalization in the monadic language $\Lambda^{\mathrm{m}}$

purely logical, hence we are not obliged to provide proofs for them. In a logic allowing inductive definitions (as in Coq), the definition of $\mathbf{N}$ can naturally be provided as transliteration of its inductive definition.

The axiomatization of normalization in $\Lambda^{\mathrm{m}}$ is presented in Figure 2.1. $\mathbf{N}$ has the meaning explained above, and similarly, $\mathbf{A}$ relates values with their normal forms, and $\mathbf{C}$ relates let-contexts with their normal forms.

Axioms 1-6 simultaneously define normal forms of expressions, values and let-contexts. Axiom 7 states that the relation $\mathbf{N}$ contains the transitive closure of one-step reductions. One-step reduction relation is itself represented by the predicate $\mathbf{Rd}$, and formalized by Axioms 8-9 which encode the two monadic reductions stated earlier. Finally, Axiom 10 defines one-step reductions in context (within the body of let-expressions).

We now introduce a logical relation for each syntactic category in the monadic language, expressing the corresponding reducibility properties in the following way:

$$\mathbf{R_v}(v) \stackrel{\mathrm{df}}{=} \exists v'.\, \mathbf{A}(v, v')$$

$$\mathbf{R_e}(e) \stackrel{\mathrm{df}}{=} \exists e'.\, \mathbf{N}(e, e')$$

$$\mathbf{R_c}(\mathtt{c}) \stackrel{\mathrm{df}}{=} \forall v.\, \mathbf{R_v}(v) \to \mathbf{R_e}(\mathtt{c}[\mathtt{ret}\, v])$$

Relations $\mathbf{R_v}$ and $\mathbf{R_e}$ state that values and expressions, respectively, normalize according to axioms from Figure 2.1. Relation $\mathbf{R_c}(\mathtt{c})$ expresses the property that a let-context $\mathtt{c}$ plugged with a normalizable value, itself also normalizes (as

an expression)[3]:

As mentioned above, instead of proving correctness for the function $\mathcal{E}'$, we prove a more general result saying that for each source term $\mathtt{t}$ there is a normal form in the target language $\Lambda^{\mathtt{m}}_{\mathtt{nf}}$ such that it satisfies the normalization property. The theorem is proved simultaneously with an auxiliary lemma; both are stated below:

**Theorem 1.** *The following hold:*

1. *For each term* $\mathtt{t} \in \Lambda$ *not containing variables* $w, w_0, w_1$, $\mathbf{R_e}(\mathcal{E}(\,\mathtt{t}\,))$ *holds (i.e., there exists a monadic normal form* $\mathtt{e} \in \Lambda^{\mathtt{m}}_{\mathtt{nf}}$ *such that* $\mathbf{N}(\mathcal{E}(\,\mathtt{t}\,), \mathtt{e})$.

2. *For each term* $\mathtt{t} \in \Lambda$ *not containing* $w, w_0, w_1$ *and for each let-context* $\mathtt{c}$ *such that* $\mathbf{R_c}(\mathtt{c})$ *holds,* $\mathbf{R_e}(\mathtt{c}[\mathcal{E}(\,\mathtt{t}\,)])$ *holds (i.e., there exists a monadic normal form* $\mathtt{e} \in \Lambda^{\mathtt{m}}_{\mathtt{nf}}$ *such that* $\mathbf{N}(\mathtt{c}[\mathcal{E}(\,\mathtt{t}\,)], \mathtt{e})$.

*Proof.* The proof proceeds by simultaneous induction on the structure of source terms.

**1: case** $\mathtt{x}$**.** We apply the definition of $\mathcal{E}$ to $\mathtt{x}$, so we have to prove $\mathbf{N}(\mathtt{ret}\,\mathtt{x}, \mathtt{e})$ for some $\mathtt{e}$. Taking $\mathtt{e} = \mathtt{ret}\,\mathtt{x}$ gives the required property by applying Axiom 3 and then Axiom 1.

**1: case** $\lambda\mathtt{x}.\mathtt{t}$**.** In this case we have to prove $\mathbf{N}(\mathtt{ret}\,(\lambda\mathtt{x}.\mathcal{E}(\,\mathtt{t}\,)), \mathtt{e})$ for some $\mathtt{e}$. By induction hypothesis 1. there exists $\mathtt{e}_0$ such that $\mathbf{N}(\mathcal{E}(\,\mathtt{t}\,), \mathtt{e}_0)$, so we can take $\mathtt{e} = \lambda\mathtt{x}.\mathtt{e}_0$, for which the claim holds by Axiom 3 and then Axiom 2.

**2: case** $\mathtt{x}$**.** Assume $\mathtt{c}$ is a let-context such that $\mathbf{R_c}(\mathtt{c})$ holds. We need to prove $\mathbf{N}(\mathtt{c}[\mathtt{ret}\,\mathtt{x}], \mathtt{e})$. This fact follows immediately by unfolding the definition of $\mathbf{R}$ since $\mathbf{A}(\mathtt{x}, \mathtt{x})$ by Axiom 1.

**2: case** $\lambda\mathtt{x}.\mathtt{t}$**.** Assume $\mathtt{c}$ is a let-context such that $\mathbf{R_c}(\mathtt{c})$ holds. We need to prove $\mathbf{N}(\mathtt{c}[\mathtt{ret}\,\lambda\mathtt{x}.\mathcal{E}(\,\mathtt{t}\,)], \mathtt{e})$. By induction hypothesis 1., we have $\mathbf{N}(\mathtt{t}, \mathtt{e})$ and hence $\mathbf{A}(\lambda\mathtt{x}.\mathcal{E}(\,\mathtt{t}\,), \lambda\mathtt{x}.\mathtt{e})$ by Axiom 2. We obtain the desired claim by unfolding the definition of $\mathbf{R}$.

**1: case** $\mathtt{t}_0\,\mathtt{t}_1$**.** We have to prove $\mathbf{N}(\mathtt{c}_0[\mathcal{E}(\,\mathtt{t}_0\,)], \mathtt{e})$ for some $\mathtt{e}$, where

$$\mathtt{c}_0 := \mathtt{let}\ w_0 = [\,]\ \mathtt{in}\ \mathtt{let}\ w_1 = \mathcal{E}(\,\mathtt{t}_1\,)\ \mathtt{in}\ w_0\,w_1\,.$$

By induction hypothesis 2., we know that there exists some $\mathtt{e}$ such that $\mathbf{N}(\mathtt{c}_0[\mathcal{E}(\,\mathtt{t}_0\,)], \mathtt{e})$, if only

$$\mathbf{R_c}(\mathtt{c}_0) \text{ holds.} \tag{1}$$

[3]This definition resembles that of a logical relation used in Tait's proof of normalization for the simply typed lambda calculus (for terms of the arrow type), whose computational content is a function that lies at the heart of the normalization-by-evaluation function as observed by Berger [4]. It is then this relation that is responsible for the higher-orderness of the program extracted from the proof also in our case. Similar relations will be defined for the CPS transformation in Section 3.

We now want to prove (1). To this end, assume $v_0$ is a value such that $\mathbf{A}(v_0, v_1)$. Unfolding the definition of $\mathbf{R_c}(c_0)$, we see that we need to find $e_0$ such that

$$\mathbf{N}(c_0[\mathtt{ret}\, v_0], e_0) \text{ holds.} \tag{2}$$

By Axiom 8, $c_0[\mathtt{ret}\, v_0]$ reduces to $c_1[\mathcal{E}(t_1)]$, where $c_1 := \mathtt{let}\ w_1 = [] \ \mathtt{in}\ v_0\, w_1$. Now we can apply induction hypothesis 2. again to obtain an expression $e$ such that $\mathbf{N}(c_1[\mathcal{E}(t_1)], e)$ holds. Then, by Axiom 7, we will have property (2) with $e$ as $e_0$. But to be able to apply the induction hypothesis, we need to show that $c_1$ satisfies property $\mathbf{R}$. So let $w_0$ be a value such that $\mathbf{A}(w_0, w_1)$ for some $w_1$. We have to find $e$ such that $\mathbf{N}(c_1[\mathtt{ret}\, w_0], e)$. Again, by Axiom 8, $c_1[\mathtt{ret}\, w_0]$ reduces to $v_0\, w_0$. Obviously, this expression normalizes to $v_1\, w_1$ (we apply Axiom 4 and then Axiom 1 twice), and by Axiom 7 we also have $\mathbf{N}(c_1[\mathtt{ret}\, w_0], v_1\, w_1)$. This concludes the proof of (2) and hence, the proof of (1).

**2: case $t_0\, t_1$.** Assume $c$ is a let-context such that $\mathbf{R_c}(c)$ holds. We need to show

$$\mathbf{R_e}(c[\mathtt{let}\ w_0 = \mathcal{E}(t_0)\ \mathtt{in}\ \mathtt{let}\ w_1 = \mathcal{E}(t_1)\ \mathtt{in}\ w_0\, w_1]).$$

First, we notice that

$$c[\mathtt{let}\ w_0 = \mathcal{E}(t_0)\ \mathtt{in}\ \mathtt{let}\ w_1 = \mathcal{E}(t_1)\ \mathtt{in}\ w_0\, w_1]$$

reduces in two steps to

$$e' \stackrel{\mathrm{df}}{=} \mathtt{let}\ w_0 = \mathcal{E}(t_0)\ \mathtt{in}\ \mathtt{let}\ w_1 = \mathcal{E}(t_1)\ \mathtt{in}\ c[w_0\, w_1],$$

according to Axiom 9. Thus it is enough to prove that $e'$ normalizes to some $e$. The proof of this fact is analogous to the proof for the previous case, except that in the end we are left with the following property to prove:

$$\mathbf{N}(c[v_0\, w_0], e) \text{ holds.} \tag{3}$$

We cannot use the fact that $\mathbf{R_c}(c)$ holds directly, since $v_0\, w_0$ is not a value. Instead, we introduce a new variable $w$ (fresh with respect to the whole proof) and we first prove the "eta-expanded" property (3), i.e., we prove that

$$\mathbf{N}(\mathtt{let}\ w = v_0\, w_0\ \mathtt{in}\ c[\mathtt{ret}\, w], e) \text{ holds.} \tag{4}$$

By assumption, $c[\mathtt{ret}\, w]$ normalizes, therefore by Axiom 6, also $\mathtt{let}\ w = v_0\, w_0\ \mathtt{in}\ c[\mathtt{ret}\, w]$ normalizes. The last problem is to prove (3) when we know (4). This can be done by induction on the length of the reduction sequence in (4). The case of length $0$ is impossible, since there is a redex in the term. Therefore (3) follows from falsity in this case. Assume the property holds for $n \geq 0$ steps. Then there are 4 possibilities for $e = \mathtt{let}\ w = v_0\, w_0\ \mathtt{in}\ c[\mathtt{ret}\, w]$ to reduce in the first step:

1. $\mathbf{Rd}(e, \texttt{let } w = v_0' \, w_0 \texttt{ in } \mathtt{c}[\texttt{ret}\, w])$ where $v_0 = \lambda x.\mathtt{t}$ and $v_0' = \lambda x.\mathtt{t}'$, with $\mathbf{Rd}(\mathtt{t}, \mathtt{t}')$. In this case also $\mathbf{Rd}(\mathtt{c}[v_0 \, w_0], \mathtt{c}[v_0' \, w_0])$, and by induction hypothesis
   $\mathtt{c}[v_0 \, w_0]$ normalizes.
2. Analogous to the previous case, but this time $w_0$ reduces.
3. If $\mathtt{c} = \texttt{let } \mathtt{x} = [] \texttt{ in } \mathtt{u}$ and $\mathbf{Rd}(\mathtt{u}, \mathtt{u}')$, then we see that $\mathbf{Rd}(e, \texttt{let } w = v_0 \, w_0 \texttt{ in } \mathtt{c}'[\texttt{ret}\, w])$ holds, where $\mathtt{c}' := \texttt{let } \mathtt{x} = [] \texttt{ in } \mathtt{u}'$. Hence $\mathbf{Rd}(\mathtt{c}[v_0 \, w_0], \mathtt{c}'[v_0 \, w_0])$ holds, and by induction hypothesis we obtain (3).
4. If $\mathtt{c} = \texttt{let } \mathtt{x} = [] \texttt{ in } \mathtt{u}$ and if

   $$\mathbf{Rd}(e, \texttt{let } w = v_0 \, w_0 \texttt{ in } \mathtt{u}[\mathtt{x} \mapsto w]),$$

   then $\texttt{let } w = v_0 \, w_0 \texttt{ in } \mathtt{u}[\mathtt{x} \mapsto w]$ normalizes in $\mathtt{n}$ steps by assumption. But by $\alpha$-renaming of the let-bound variable $w$, $\texttt{let } w = v_0 \, w_0 \texttt{ in } \mathtt{u}[\mathtt{x} \mapsto w] =_\alpha \mathtt{c}[v_0 \, w_0]$, hence we conclude (3).

□

In the last case of the proof, we took care not to peek into the structure of the let-context $\mathtt{c}$ so that it can represent the continuation (functional accumulator) in the extracted program. As can be seen, we can still prove the required property, but we need some additional tools: the induction principle for the normalization predicate $\mathbf{N}$ and the inversion principle for the reduction relation $\mathbf{Rd}$. In Coq, the two properties are provided by the system, provided we define both predicates inductively. In first-order logic, we would have to add additional axioms ensuring these properties. This addition does not pose any problems if we use the induction principle for non-computational properties (as it is in our case) – then the axiom does not carry computational content either. The inversion axioms for reduction are also purely logical. Lastly, we also need to use an axiom for alpha-renaming of let-bound variables to complete the proof. Have we chosen to explicitly use the structure of $\mathtt{c}$, we would have to formalize substitution and we would end up in a substitution-based normalization function, whereas now we avoid dealing with substitution altogether.

## 2.2 The extracted program

Let us discuss the computational content of the proof of Theorem 1. Informally, a proof in constructive logic can be encoded as a term in some language of terms corresponding to this logic by the Curry-Howard isomorphism. Such a term typically contains a logical part (encoding proofs of properties) and a computational part (encoding construction of objects). The extraction of a program from the proof consists in erasing the logical part and keeping the computational part that can then be seen as implementing an algorithm "hidden" in the method of constructive proof. One of the simple settings is first-order intuitionistic logic

where the construction of an object is done by constructing witnesses for existential quantifiers. Formally, the extraction can be obtained by using Kreisel's modified realizability proof interpretation. This approach can be applied also in our case, except that we have to do induction on terms in Theorem 1 at the meta-level, yielding a family of programs for each source term rather than a single program acting on terms. An optimization of modified realizability due to Berger gives exactly the programs that we started with. The development follows the lines of those in Berger [4] or of Biernacka et al. [7]. The main idea is that proof terms of a proposition $\mathsf{P}$ give rise to typed programs. The type of program is determined based on the structure of the proposition according to the following definition:

$$\begin{array}{lcl} \tau(\mathcal{A}(\mathsf{t}_1,\ldots,\mathsf{t}_n)) & := & () \\ \tau(\phi \to \psi) & := & \tau(\phi) \to \tau(\psi) \\ \tau(\forall\, \mathsf{x}^{\iota}.\, \phi) & := & \iota \to \tau(\phi) \\ \tau(\exists\, \mathsf{x}^{\iota}.\, \phi) & := & \iota \times \tau(\phi) \end{array}$$

In particular, the type of the program extracted from Theorem 1 is the following: $\mathtt{term} \to \mathtt{mon}_{\mathtt{nf}}$, where $\mathtt{term}$ is the type of lambda terms (corresponding to the logical sort $\Lambda$) and $\mathtt{mon}_{\mathtt{nf}}$ is the type of monadic normal forms (corresponding to the logical sort $\Lambda^{\mathsf{m}}$) in the extracted program. The type of the logical relation $\mathbf{R}$ is $\mathtt{val} \to \mathtt{val}_{\mathtt{nf}} \to \mathtt{mon}_{\mathtt{nf}}$, where $\mathtt{val}$ is the type of monadic values and $\mathtt{val}_{\mathtt{nf}}$ is the type of monadic normal values, and it becomes the functional accumulator. A crucial observation is that the first argument of type $\mathtt{val}$ of this functional accumulator can be eliminated as dead code, because it is not used in the function to construct the final value; only the second argument is used, and the first one appears in the extracted code as an artefact of the universal quantifier in the definition of $\mathbf{R}$. The logical foundation for dead code elimination of this kind is an annotation of the proof, where we mark all those quantifiers whose instantiations do not occur in witnesses for the computationally relevant quantifiers. That is to say, we can introduce a new kind of "marked" quantifiers whose type ignores the argument:

$$\tau(\{\forall \mathsf{x}^{\iota}\}.\, \phi) := \tau(\phi)$$

provided we make sure that $\mathsf{x}$ is computationally redundant, i.e., it does not occur free in any witness within $\phi$. Furthermore, since we erase the type argument to functions corresponding to such marked quantifiers, we are also able to erase witnesses (i.e., arguments provided) to such marked quantifiers (i.e., extracted functions) in a logically sound way (this optimization has been shown to be sound by Berger [4, 5]). As a consequence, we are able to eliminate all the expressions in the intermediate monadic language, or in other words to perform deforestation in the extracted function.

Let us now discuss the main differences of this approach with our development in Coq, carried out fully formally. First, the logic underlying Coq is higher-order and it supports inductive definitions, therefore all the axioms

assumed in Section 2.1 can be implemented naturally following the inductive definitions of one-step reduction and normalization. Moreover, the induction axiom on **N** and the inversion property on **Rd** used in the end of the proof can then be automatically generated by the Coq system, based on their inductive specification.

**Variable names and freshness.** Mechanized support for representing and proving properties of languages with binding is currently an active area of study with no clear solution as yet, at least in the Coq community[4], so we decide to steer clear of this discussion in order not to obscure the picture. In the Coq development, we assume an abstract data type of names (`name`) and three distinguished variables of this type ($w, w_0, w_1$ as used in the proof in Section 2.1). We only require decidability of equality on type `name` and its actual implementation is delegated to the user of the extracted program (the abstract type and variables are left unspecified and are indicated as "axioms to be realized" in the extracted code). An alternative would be to use de Bruijn indices for variables, or a mixed representation using names for free variables and de Bruijn indices for bound variables [18]. In the latter case, it is also up to the user of the extracted code to instantiate the abstract type of names with a specific type, but it is not required to satisfy any conditions in order to maintain correctness of the extracted code.

**The extraction mechanism in Coq.** In Coq, all objects are declared by the user to be either computational, or logical by declaring the right sort (`Set` or `Prop`, respectively). An object of the former kind is extracted to a functional program in Ocaml, Haskell or Scheme (in a way significantly more complicated than explained above in the setting of first-order logic, but the idea is the same), and an object of the latter kind is computationally irrelevant, so it is erased during extraction. In particular, inductively defined relations are logical and inductively defined data types are computational. Regarding the formalization of the proof given in Section 2.1, we define the following objects and their sorts:

- data types of lambda terms, monadic terms are computational
- relations $\mathbf{N}, \mathbf{A}, \mathbf{C}$ are logical
- the axioms we assume in the development are logical (hence, their proofs are not needed for the construction of the program we want to obtain)
- theorems can be either logical or computational; in our case, e.g., Theorem 1 is computational: for each term, we prove the (constructive) existence of a monadic normal form, therefore its computational content will be a function mapping terms to monadic normal forms. The computational counterpart of a logical existential quantifier is a dependent sum (cf. the Coq code).

[4] See, for example, the preliminary work of Aydemir et al. on automatic support for languages with binders in Coq [2] or Leroy's solution to POPLmark challenge [18].

```
type name = ... (* to be realized *)
let w = ... (* to be realized *)
let w0 = ... (* to be realized *)
let w1 = ... (* to be realized *)

(* source terms *)
type term =
  | Var of name
  | Lam of name * term
  | App of term * term

(* monadic normal forms *)
type mon_nf =
  | Ret of val_nf
  | Let of cont_nf * val_nf * val_nf
  | V_app of val_nf * val_nf
and val_nf =
  | Mvar of name
  | Mlam of name * mon_nf
and cont_nf =
  | Cont of name * mon_nf

(* norm : term -> mon_nf *)
let rec norm t =
  match t with
    | Var x -> Ret (Mvar x)
    | Lam (x,t) -> Ret (Mlam (x, norm t))
    | App (t0,t1) ->
          normc t0
            (fun v ->
               normc t1
                 (fun v0 -> V_app (v, v0)))

(* normc : term -> (val_nf -> mon_nf) -> mon_nf *)
and normc t c =
  match t with
    | Var x -> c (Mvar x)
    | Lam (x,t) -> c (Mlam (x, norm t))
    | App (t0,t1) ->
        normc t0
          (fun v ->
            normc t1
              (fun v0 ->
                Let ((Cont (w, (c (Mvar w)))), v, v0)))
```

Figure 2: A higher-order, one-pass transformation into monadic normal forms

In general, Coq objects may be dependently typed and their extraction to non-dependently typed target languages can be messy, and it may involve unsafe coercions to ensure typing. Fortunately, this is not the case in our development and the program we extracted is quite compact and clean.

**Axiom 11.** $\forall x.\, \mathbf{A}(x, x)$

**Axiom 12.** $\forall x e_0 e_1.\, \mathbf{N}(e_0, e_1) \to \mathbf{A}(\lambda x.e_0, \lambda x.e_1)$

**Axiom 13.** $\forall v_0 v_1.\, \mathbf{A}(v_0, v_1) \to \mathbf{N}(\kappa\, v_0, \kappa\, v_1)$

**Axiom 14.** $\forall v_0 w_0 v_1 w_1.\, \mathbf{A}(v_0, v_1) \to \mathbf{A}(w_0, w_1) \to \mathbf{C}(c_0, c_1) \to \mathbf{N}(v_0\, w_0\, c_0, v_1\, w_1\, c_1)$

**Axiom 15.** $\forall \kappa.\, \mathbf{C}(\kappa, \kappa)$

**Axiom 16.** $\forall w e_0 e_1.\, \mathbf{N}(e_0, e_1) \to \mathbf{C}(w\, e_0, w\, e_1)$

**Axiom 17.** $\forall \kappa e_0 e_1.\, \mathbf{N}(e_0, e_1) \to \mathbf{N}_r(\lambda\kappa.e_0, \lambda\kappa.e_1)$

**Axiom 18.** $\forall \kappa e c.\, \mathbf{Rd}((\lambda\kappa.e)\, c, e[\kappa \mapsto c])$

**Axiom 19.** $\forall v w e.\, \mathbf{Rd}((\lambda v.e)\, w, e[v \mapsto w])$

**Axiom 20.** $\forall e_0 e_1 e_2.\, \mathbf{Rd}(e_0, e_1) \to \mathbf{N}(e_1, e_2) \to \mathbf{N}(e_0, e_2)$

Figure 3: Axiomatization of normalization of CPS terms

A closer inspection of the proof in Section 2.1 reveals that the computational content of Theorem 1 is a pair of mutually recursive functions: one that maps source terms directly to monadic normal forms, and the other that maps a source term and a context to a normal form corresponding to the monadic term obtained by plugging the encoding of the source term into the context. The fact that the program is higher-order arises from the use of the relation $\mathbf{R}$ and it operates in one pass because we do not rely on constructing intermediate monadic terms (images of the encoding). The proof itself is straightforward, except for the last case and the proof of property (4): in order to avoid peeking into the structure of the let-context $c$ (so that we can eliminate it as dead code at extraction time), we needed to introduce a form of eta expansion in order to be able to reorganize let-constructs. This construction is responsible for an eta-expanded use of the continuation $\kappa$ in the last line of Danvy's transformation.

The Coq extraction mechanism does not allow to mark computational fragments of the proof as irrelevant so they can be erased during extraction time, as in optimized modified realizability interpretation. Such a need arises in the present case: in the proof of Theorem 1, for each source term, we do not only construct its monadic normal form, but also intermediate monadic values or contexts as defined by the naive encoding. However, this intermediate term does not take part in constructing the target normal form. Using modified realizability, we could identify and annotate such occurrences, but not in Coq, where the computational relevance is decided by the sort (`Prop` or `Set`) once and for all. However, we can eliminate the intermediate terms by hand (simply by erasing function arguments which are not used), obtaining an implementing

Danvy's one-pass transformation $\mathcal{E}'$. It is shown in Figure 2.

One thing worth noting here is that the program extracted is not presented using two top-level, mutually recursive functions as in Figure 2 due to the fact that in Coq we do not prove two cases of Theorem 1 independently, but at the same time, i.e., using a single induction principle on source term to prove a conjunction of two properties. Therefore, the extracted program is a function from term to pairs. It is a matter of untangling this definition to state in the (more readable) form as in Figure 2.

The formalization and the proof is relatively simple as regards the logic involved, hence it should pose no problems to migrate the development to other proof assistants.

# 3 A higher-order, one-pass CPS transformation

In this section, we sketch a similar development for a higher-order, one-pass call-by-value transformation of lambda terms into continuation-passing style [1, 9, 10, 28] (a similar development for the call-by-name case can be carried out in an analogous way). The general method and structure of the proof is similar to that of Section 2, and so is the structure of the program extracted from the proof.

In case of the CPS transformation the intermediate language is that of CPS terms as defined by Plotkin's original (non-optimizing) translation [23] and the target language is that of "compact" CPS terms, i.e., with the administrative β-redexes reduced away. Thus the grammar of intermediate CPS language (a subset of ordinary lambda terms) is the following:

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

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

$$\begin{aligned}
&\overline{\mathsf{x}} = \lambda\kappa.\kappa\,\mathsf{x} \\
&\overline{\lambda\mathsf{x}.\mathsf{t}} = \lambda\kappa.\kappa\,(\lambda\mathsf{x}.\overline{\mathsf{t}}) \\
&\overline{\mathsf{t}_0\,\mathsf{t}_1} = \lambda\kappa.\overline{\mathsf{t}_0}(\lambda\mathsf{f}.\overline{\mathsf{t}_1}(\lambda v.\mathsf{f}\,v\,\kappa))
\end{aligned}$$

The "administrative redexes" in CPS-translated terms are β-redexes introduced at translation time and not present in the source term. For example, ...

An administrative redex is either an application of a CPS-translated subterm to a continuation, or an application of a continuation to a value. A residual β-redex is characterized as an application of a value to a value. Therefore, terms

in normal form with respect to administrative redexes (compact CPS terms) can be defined using the following grammar:

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

Again, similarly to the monadic case, we formalize the process of normalization of intermediate CPS terms into compact CPS terms. Recall that in the lambda calculus in general, normalization does not always terminate, and it can be proven to do so if we restrict ourselves to certain classes of typed or typable terms [16]. In the simply typed lambda calculus, for example, we can prove normalization using a family of reducibility predicates indexed by types: we define a reducibility property strong enough to entail normalization, and we subsequently show that all well-typed terms have that property. In the present case, we define a similar notion of reducibilty predicate for lambda terms, except that they are not type-directed (in fact, they need not be typed) but syntax-directed: the restricted syntax of intermediate CPS terms ensures that it is indeed possible to normalize all such terms. We define four predicates, one for each syntactic category of CPS terms, expressions, continuations and values.

Before stating these predicates, let us first choose atomic predicates needed for the axiomatization of normalization. Analogously to the monadic language, we take the following: $\mathbf{N}(r_0, r_1)$ is a logical predicate meaning that a term $r_0$ normalizes to $r_1$; similarly, $\mathbf{N}(e_0, e_1)$ relates expressions and normal expressions, $\mathbf{A}(v_0, v_1)$ relates values and normal values, and $\mathbf{C}(c_0, c_1)$ relates continuations with normal continuations. All the syntactic categories are plain lambda terms, so normalization means the usual $\beta$-normalization but applied statically.

The axiomatization of normalization in the intermediate CPS language is presented in Figure 3. Axioms 11-12 characterize normal values, Axioms 13-14 characterize normal expressions, Axioms 15-16 characterize normal continuations, and Axiom 17 defines a normal CPS term. Finally, Axioms 18-19 define one-step reduction ($\beta$-reduction) of the two kinds explained above, and Axiom 20 states that $\mathbf{N}$ contains the transitive closure of one-step reduction. Note that only expressions can be reduced.

We can now return to the reducibility predicates, i.e., relations on all syntactic categories defining what it means for each syntactic construct to be "reducible":

$$\begin{aligned} \mathbf{R_e}(e_0) &\stackrel{\text{df}}{=} \exists\, e_1.\, \mathbf{N}(e_0, e_1) \\ \mathbf{R_v}(v_0) &\stackrel{\text{df}}{=} \exists\, v_1.\, \mathbf{A}(v_0, v_1) \\ \mathbf{R_c}(c) &\stackrel{\text{df}}{=} \forall\, v.\, \mathbf{R_v}(v) \;\rightarrow\; \mathbf{R_e}(c\, v) \\ \mathbf{R_r}(r) &\stackrel{\text{df}}{=} \forall\, c.\, \mathbf{R_c}(c) \;\rightarrow\; \mathbf{R_e}(r\, c) \end{aligned}$$

The base cases are reducibility of expressions ($\mathbf{R_e}$) and values ($\mathbf{R_v}$) in which cases we require normalization directly. 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.

We now state the theorem about the relation between source lambda terms and normal forms in the intermediate CPS language, similarly to the monadic case.

**Theorem 2.** *For each term* $\mathtt{t} \in \Lambda$*, there exists a compact CPS term* $\mathtt{r}$ *(a term in CPS-normal form) such that* $\mathbf{N}_\mathtt{r}(\bar{\mathtt{t}}, \mathtt{r})$.

Not surprisingly, the proof follows the idea and structure of the normalization proof for the simply typed lambda calculus using Tait's reducibility predicates [16, 20]. It uses two auxiliary lemmas:

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

**Lemma 2.** *For each term* $\mathtt{r}$ *such that* $\mathbf{R_r}(\mathtt{r})$, $\mathbf{N}_\mathtt{r}(\mathtt{r}, \mathtt{r}')$ *holds for some* $\mathtt{r}'$ *(CPS normal form).*

Lemma 1 shows that the CPS encoding of all source terms has the property $\mathbf{R_r}$, and by Lemma 2 we then deduce that all terms have CPS normal forms (i.e., the corresponding "compact" CPS terms).

The proof of Lemma 1 proceeds by induction on the structure of source terms and, just as in the monadic case, the construction of the witness for the existential quantifier in each case explicitly avoids using the intermediate translation $\bar{\cdot}$. Also, the proof can be constructed in such a way that we do not depend on the syntactic structure of the continuation $\mathtt{c}$ used in the predicate $\mathbf{R_r}$, hence we do not have to deal with substitution and in the extracted program the computational content of $\mathtt{c}$ is a proper continuation (eta-expanded when needed).

In both cases, we use the fact that reduction preserves the reducibility property "backwards", i.e., the following lemma holds:

**Lemma 3.** *For every expression* $\mathtt{e}$*, if* $\mathtt{e}$ *reduces to* $\mathtt{e}'$ *and* $\mathbf{R_e}(\mathtt{e}')$ *holds, then also* $\mathbf{R_e}(\mathtt{e})$ *holds.*

The computational content of $\mathbf{R_e}$ are normal expressions (witnesses for the existential quantifier), similarly proof terms of $\mathbf{R_v}$ are extracted to normal values. The computational content of $\mathbf{R_c}$ is a function of type $\mathtt{val} \to \mathtt{val}_{\mathtt{nf}} \to \mathtt{exp}_{\mathtt{nf}}$ where $\mathtt{val}$ and $\mathtt{val}_{\mathtt{nf}}$ are the types of CPS values and normal values, respectively, and $\mathtt{exp}_{\mathtt{nf}}$ is the type of normal expressions. Again, an analysis similar to that of Section 2.2 leads to the conclusion that the first argument to this function is not used, or – equivalently – it is computationally redundant in the constructed proof, so it can be eliminated at extraction time. Thus we are left with the program presented in Figure 4.

Similarly to the monadic case, the program extracted from the proof of

Theorem 2 coincides with the known higher-order, one-pass transformation:

$$\begin{aligned}
&\mathcal{C}\,(x\,,\kappa) = \kappa@x\\
&\mathcal{C}\,(\lambda x.t\,,\kappa) = \kappa@(\lambda x.\lambda\kappa'.\mathcal{C}\,(t\,,\overline{\lambda}u.\kappa'\,u))\\
&\qquad \text{where } \kappa' \text{ is fresh}\\
&\mathcal{C}\,(t_0\,t_1\,,\kappa) =\\
&\qquad \mathcal{C}\,(t_0\,,\overline{\lambda}v_0.\mathcal{C}\,(t_1\,,\overline{\lambda}v_1.v_0\,v_1\,(\lambda w.\kappa@w)))\\
&\qquad \text{where } w \text{ is fresh}
\end{aligned}$$

At top level, the one-pass transformation uses $\mathcal{C}$ in the following way:

$$\mathcal{C}_t\,t = \lambda\kappa.\mathcal{C}\,(t\,,\overline{\lambda}u.\kappa\,u),$$

where $\kappa$ is chosen fresh.

The program in Ocaml is presented in Figure 4. We use a `gensym` function to generate fresh names, and we can simulate this fact in the Coq development for example by assuming a supply of fresh variable names with respect to a given set of names.

# 4 Conclusion

We have proved correctness of two known higher-order, one-pass transformations from the lambda calculus to monadic normal forms and to compact CPS terms by extracting them from proofs of suitable existence properties in a logic. Thereby we have also shown a general approach to both proving correctness of existing programs (one can craft the proof to obtain the required extracted code) and to obtaining new, provably correct, one-pass transformations given a non-optimizing translation and a specification of normalizability in the target language as a rewriting system. We argue that the presented method of writing these particular higher-order programs may in some cases be easier than having to write a function from scratch, since the ingredients needed in our approach are often well known for a given language, and it is a matter of putting the pieces of the puzzle together to mechanically obtain the code, hopefully using an automated proof assistant to help in the development. We also note the that our programs are in each case extensionally equivalent to the composition of the naive translation with the reduction-free normalization function for the meta-language (i.e., instances of normalization by evaluation). This fact is in analogy to the similar observation made for the normalization proof of the simply typed lambda calculus using logical relations, whose computational content has been shown to be the first instance of NbE.

Last but not least, we would also like to advocate this "logic-driven" approach to code generation, especially when its provable correctness is crucial: with a little of experience, one can view developing proofs in Coq (or a similar system) as "certified" programming; in particular, one can adjust the proof method to obtain the desired intensional structure of the code.

```
(* source terms *)
type term =
  | Var of name
  | Lam of name * term
  | App of term * term

(* target cps terms *)
type cps_tm =
  | Tt of name * exp
and exp =
  | Te_cont of name * tval
  | Te_app of tval * tval * name * exp
and tval =
  | Tvar of name
  | Tlam of name * cps_tm

(* transformation *)
(* cpsopt : cps_tm -> (tval -> cps_tm) -> cps_tm *)
let rec cpsopt t k =
  match t with
    | Var n -> k (Tvar n)
    | Lam (n,t) -> let k' = gensym "k"
                   in
                     k (Tlam (n, (Tt (k', cpsopt t (fun v -> Te_cont (k', v))))))
    | App (t0,t1) ->
        let u = gensym "u"
        in cpsopt t0
             (fun v ->
                cpsopt t1
                  (fun v0 -> Te_app (v, v0, u, k (Tvar u))))

(* invocation at the top level, with the identity continuation *)
(* top : term -> cps_tm *)
let top t = let k = gensym "k"
            in
              Tt (k, cpsopt t (fun u -> Te_cont (k, u)))
```

Figure 4: A higher-order, one-pass call-by-value transformation to compact CPS terms

# 5 Acknowledgments

This work has been supported by the MNiSW grant number N N206 357436, 2009-2011.

# References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[2] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in coq. *Electronic Notes in Theoretical Computer Science*, 174(5):69–77, 2007. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2007.01.028.

[3] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java byte-codes. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 1, pages 129–140, Baltimore, Maryland, September 1998. ACM Press.

[4] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[5] Ulrich Berger. Uniform Heyting Arithmetic. *Annals of Pure and Applied Logic*, 133:125–148, 2005.

[6] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82, 2005. Special issue.

[7] Małgorzata Biernacka, Olivier Danvy, and Kristian Støvring. Program extraction from proofs of weak head normalization. In Martin Escardó, Achim Jung, and Michael Mislove, editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics(MFPS XXI)*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 169–189, Birmingham, UK, May 2005. Elsevier Science Publishers. Extended version available as the research report BRICS RS-05-12.

[8] Olivier Danvy. A new one-pass transformation into monadic normal form. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 77–89, Warsaw, Poland, April 2003. Springer-Verlag.

[9] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

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

[11] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002*, number 2303 in Lecture Notes in Computer Science, pages 98–113, Grenoble, France, April 2002. Springer-Verlag.

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

[13] Zaynah Dargaye and Xavier Leroy. Mechanized verification of cps transformations. pages 211–225. 2007. doi: 10.1007/978-3-540-75560-9_17. URL `http://dx.doi.org/10.1007\%2F978-3-540-75560-9\_17`.

[14] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[15] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations (with retrospective). In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979–1999, A Selection*, pages 502–514. ACM Press, 2004.

[16] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[17] Andrew Kennedy. Compiling with continuations, continued. *SIGPLAN Not.*, 42(9):177–190, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291179.

[18] Xavier Leroy. A locally nameless solution to the POPLmark challenge. Research report 6098, INRIA, January 2007. URL `http://gallium.inria.fr/~xleroy/publi/POPLmark-locally-nameless.pdf`.

[19] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[20] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic*

*Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

[21] Yasuhiko Minamide and Koji Okuma. Verifying cps transformations in isabelle/hol. In *MERLIN '03: Proceedings of the 2003 ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 1–8, New York, NY, USA, 2003. ACM. ISBN 1-58113-800-8. doi: http://doi.acm.org/10.1145/976571.976576.

[22] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

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

[24] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. pages 289–360, 1993. A preliminary version was presented at the 1992 ACM Conference on Lisp and Functional Programming (LFP 1992).

[25] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997. A preliminary version was presented at the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP 1996).

[26] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.1.* Available at `http://coq.inria.fr/V8.1pl3/refman/index.html`.

[27] Ye Henry Tian. Mechanically verifying correctness of cps compilation. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 41–51, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. ISBN 1-920682-33-3.

[28] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 294–311, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag.