Typeful Continuations

Matthias Puech

Typeful Continuations

Matthias Puech[1]

[1]: *Inria and LIX, École Polytechnique, France*
matthias.puech@inria.fr

Abstract

Continuation-passing style translations, or CPS, are used notably in compilers. They make a functional program more explicit by sequentializing its computations and reifying its control. They have been used as an intermediate language in many compilers. Yet, we cannot directly use Plotkin's original CPS translation to design compilers, for several reasons: *i)* the resulting terms it produces are verbose (they contain "administrative redexes", a well-studied problem) and inefficient (they perform nested beta-reductions sequentially); and *ii)* their loose syntax can hide possible further optimizations, in particular tail call elimination.

These issues have been solved in isolation and on paper, but never combined together and formalized, which *i)* is necessary to scale to safe, full-featured optimizing compilers and *ii)* is not a mere juxtaposition of the techniques but requires insight on the design of CPS translations themselves. We propose to design and implement in OCaml a compacting, optimizing CPS translation, while using OCaml's type system to verify that it maps well-typed terms to well-typed terms in a tightly restricted syntactical form (the "typeful" approach to formalization). The resulting artifact can be used directly to design optimizing compilers, and the path towards it offers a tutorial on optimizing CPS translations, and yet another use case for OCaml's advanced type system.

1. Introduction

How should we design compilers so as to formally guarantee that the output code will behave the same way as the input one should? This vast research project, started some 30 years ago, comes today to fruition in particular for low-level languages like C [15]. An ongoing effort aims at extending it to higher-level ones, in particular functional languages, with the λ-calculus at their core [10].

Continuation-passing style Among the compilation schemes for functional languages, Continuation-Passing Style, or CPS, is a popular choice [2]. CPS is a classic and powerful tool that has seen several decades of study and applications, in very diverse fields: functional programming, semantics, compiler design, proof theory... It notably transforms a functional program, potentially triggering non-local jumps (e.g., using Lisp's `call/cc` or delimited control operators), into a pure, higher-order program which makes the evaluation strategy explicit by sequentializing its computations.

Several features leave an important gap between the theoretical transform originally posited by Plotkin [16] and the actual implementations in compilers:

- first, the original transform generated far more redexes than initially present in the source term, the so-called "administrative redexes", as an artifact of the transformation itself. This well-studied problem led to the discovery of the one-pass CPS transforms [6].
- even without administrative redexes, the output is still not optimized; in particular, it translated beta-redexes in a verbose way: transforming the nested beta-redex $(\lambda x_1 x_2 x_3.\, M)\, N_1\, N_2\, N_3$ leads to unnecessary threading of continuations [8], which requires six reduction steps instead of three.

- CPS-translation is merely one of several compilation passes; after it, one might want to perform further optimizations based on recognizing patterns in the program text. Without special care, CPS could hide these patterns and make optimizations harder. Take for instance tail call elimination: a tail call is recognizable in the source text, but might not be less recognizable in standard call-by-value CPS form.

These issues were addressed separately and on paper in the past, by designing ad-hoc optimizations to the historical transform. Unfortunately: *i)* they often required complex machinery, with no formal guarantee that they fulfilled their initial goal; *ii)* combining these optimizations in a single pass is non-trivial and requires insight on their design and on the nature of CPS terms.

In this paper, we address these together for the first time, moreover with special care on formal guarantees of type safety and fine analysis of the syntactic forms. Besides the formalization effort, this study will provide insights on the original CPS transform, and a general methodology to analyze higher-order program transformations.

We start here with Plotkin's standard call-by-value transform[16]. A similar exercise could be carried in call-by-name, which we leave as exercise to the reader.

Typeful programming We call *typeful* the style of light formalization of type preservation of program translations we adopt here; this research program was already initiated in the study of Normalization by Evaluation [7]. We first assign simple types to our source language. We then identify the tightest syntax to fit the target of the translation, so as to guarantee that the output conforms to the specification. We then code the translation itself, which gives us a type assignment for the target language.

Contrarily to the standard practice of using a full-blown proof assistant, e.g., Coq or Agda, and formalize operational semantics preservation, we carry it out in the minimal setting of a general-purpose programming language — here OCaml — with *Generalized Abstract Data Types* (GADTs for short) [11], which are a generalization of ML algebraic data types that allows a fine control on the return type of their constructors [20]. We use them to intrinsically represent simply typed terms of the various calculi, and to relate them through translation functions. They are sufficiently expressive to prove type preservation, and yet lightweight enough to be almost transparent to the programmer, as they enable a gradual approach to formalization: one can start by specifying syntax trees (abstract types) and transformations (functions), and later decorate them with object types, leaving the code unchanged. Beyond showcasing the expressive power of GADTs, we argue that this choice also shows that the gap between functional languages and proof assistants is shrinking.

Yet of course, OCaml does not provide the same safety guarantees as proof assistants; for instance, potential use of side-effects could compromise the safety of the results; we will have to be careful using them in a controlled fashion. Besides, in order to avoid the pitfall of variable binding, we use weak Higher-Order Abstract Syntax (wHOAS) to represent binders [5]. These choices, which are theoretically unsafe in our case (see discussion Section 2), would not be possible in a proof assistant based on type theory, but they lighten the development greatly.

Outline The remainder of this article is an incremental, literate programming exposition of our implementation, in OCaml, of a typeful, one-pass, beta-normal, properly tail-recursive CPS transform..[1] After briefly recalling GADTs in OCaml and Plotkin's call-by-value CPS transform (Section 2), we begin its gradual optimization. Starting from Danvy & Fillinski's one-pass CPS transform, we first exhibit its precise target syntax and typing (Section 3). The resulting system has no administrative redexes, and we show how to evaluate it. We then exhibit the inefficiency of

[1] We will however allow ourselves to reorder code snippets for pedagogical purposes. All the code presented here is available from the author's webpage.

translating multiple nested beta-redexes, show how to constrain the syntax to forbid this inefficiency, and modify the translation accordingly (Section 4). Finally, we show that the translation of tail calls gives rise to η-redexes which could be avoided (Section 5); again, we show how to constrain the syntax to forbid them, and modify the translation accordingly.

2. Background

2.1. The simply-typed λ-calculus with GADTs and HOAS

The starting point of our development is the simply-typed λ-calculus, equipped with a **let** construct for convenience.

$$M ::= \lambda x.\, M \mid M\, M \mid x \mid \mathbf{let}\, x = M\, \mathbf{in}\, M \qquad \text{Expressions}$$

We leave out its typing rules, which are absolutely standard and can be found in any respectable textbook [12]. How should we encode this language in OCaml so that only well-typed terms, and function manipulating well-typed terms, are accepted?

First of all, we will use a deep embedding of the language: each expression of our object language (the λ-calculus) will be represented by a value—more precisely an abstract data type—in the meta-language (OCaml); this is essential because we need structural recursion on its structure. Secondly, we will use Higher-Order Abstract Syntax (HOAS) to encode variable binding. When it comes to representing abstract syntax which contains binders and variables, one must ensure that only well-bound terms will be representable, and the others rejected; many techniques are in use, including the well-known De Bruijn indices. HOAS is the most concise we know and the closest to pen-and-paper practices; it boils down to using the meta-language binding capabilities to represent the object-language binders; in our case, it means using OCaml functions "**fun** `x` $\rightarrow$" to introduce a variable named x in an encoded expression, and use "`x`" to refer to it in the body of the binder. Because of this confusion between function space and binding space, HOAS problematically lets us represent values which are not terms of our object language: the so-called "exotic terms"[19]. Weak, or parametric HOAS [5] rule out these exotic terms by leaving the type of the variables abstract; this is what we will use here.

Thirdly, we will use Generalized Abstract Data Types to encode the object type system (simple types). The recent introduction of GADTs into OCaml [11] makes it possible to declare abstract data types which are indexed by types: each constructor can fix the type parameters of its type.

All in all, the following snippet introduces the OCaml type of simply-typed λ-terms:

```
type α m =
  | MLam : (α x → β m) → (α → β) m
  | MApp : (α → β) m * α m → β m
  | MVar : α x → α m
  | MLet : α m * (α x → γ m) → γ m
```

The benefit of using GADTs is that object typing appear as simple type annotations on top the usual untyped abstract data type; whereas other techniques would draw the encoding further away from it [3]. As promised, the type of variables α `x` is separated from the type of terms α `m`. Think of this type of variables as abstract for the moment; it will be defined in Section 3.3 when discussing evaluation. For instance, the applicator term $\lambda f x.\, f\, x$ is encoded as the following OCaml value:

```
let ex : α β. ((α → β) → α → β) m =
  MLam (fun f → MLam (fun x → MApp (MVar f, MVar x)))
```

Note that only well-typed λ-terms are well-typed OCaml value of type α `m`. This is characteristic of *intrinsically-typed* representations.

$$[\![x]\!]\ K = K[x] \tag{4}$$
$$[\![\lambda x.\, M]\!]\ K = K[\lambda x k.\, [\![M]\!][\lambda M.\, [k\ M]]] \tag{5}$$
$$[\![M\ N]\!]\ K = [\![M]\!]\ [\lambda M.\, [\![N]\!]\ (\lambda N.\, M\ N\ (\lambda v.\, K[v]))] \tag{6}$$
$$[\![\mathbf{let}\, x = M\, \mathbf{in}\, N]\!]\ K = [\![M]\!]\ [\lambda M.\, \mathbf{let}\, x = M\, \mathbf{in}\, [\![N]\!][\lambda N.\, K[N]]] \tag{7}$$
$$[\![M]\!] = \lambda k.\, [\![M]\!]\ [\lambda M.\, k\ M] \tag{8}$$

Figure 1: One-pass CPS-transformation

2.2. Call-by-value CPS

In its seminal 1975 article [16], Plotkin proposed a pair of source-to-source program transformations on the λ-calculus that has two important and since then famous properties: *simulation*, i.e. transformed terms reduce to similar values as source terms, and *indifference*, i.e. the transformed terms reduce the same way in all reduction strategy. One transformation simulated the call-by-name strategy, the other the call-by-value strategy. In other words, the evaluation strategy, implicit in the source language, can be made explicit by CPS-translating its programs; it turns out that many control operators and non-local stack manipulations can also be simulated by CPS [6]. This is why CPS is a form of compilation: it turns implicit notions, control and evaluation order, into an explicit one, continuation.

Here is the call-by-value transformation:

$$[\![x]\!] = \lambda k.\, k\ x \tag{1}$$
$$[\![\lambda x.\, M]\!] = \lambda k.\, k\ (\lambda x.\, [\![M]\!]) \tag{2}$$
$$[\![M\ N]\!] = \lambda k.\, [\![M]\!](\lambda m.\, [\![N]\!](\lambda n.\, m\ n\ k)) \tag{3}$$

Each subterm is transformed into a functional $(\lambda k.\ \dots)$ that awaits a *continuation* k, i.e., a function to which the result of the computation will be passed. A variable x becomes a term that passes x to its continuation; an abstraction becomes a term that passes to its continuation the abstraction body; an application becomes a term that passes sequentially to its subterms continuations $(\lambda m.\ \dots)$ and $(\lambda n.\ \dots)$, eventually forming an application $m\,n\,k$ with the results.

It should be therefore clear that the transformation has an effect on the types of the transformed term. for instance, if $M : p \supset q$, then $[\![M]\!] : ((p \supset (q \supset o) \supset o) \supset o) \supset o$ for any given *answer type* o. For this reason, directly transliterating the function $[\![\cdot]\!]$ above as an OCaml function of the GADT α `m` would lead to an ill-typed program: inferring the type transformation is beyond the scope of GADTs.

3. One-pass CPS-transformation

Plotkin's original transformation had one major drawback: it generated far more redexes than present in the source term; reducing the additional "administrative" redexes on-the-fly became an interesting challenge, solved independently by Appel, Danvy & Filinski and Wand [2, 6, 18]. By a control flow analysis of the original translation, one can classify lambdas and applications as either static, i.e., invariably forming redexes and therefore reducible at translation-time, or dynamic, i.e., to leave in the output term to be eventually reduced at run-time [6]. The one-pass CPS transform then uses the meta-language's function space to represent static redexes, leading to a higher-order transformation. In this section, we use the higher-order nature of the one-pass CPS to derive the syntax and typing of the target language of CPS.

Notations Throughout this paper, we are going to define translations and evaluations of variants of the λ-calculus using a higher-order meta-language, itself akin to a λ-calculus. To distinguish this meta-language from the object languages, we will write respectively $\lambda X.\, E$, $E_1[E_2]$ and X for the abstraction, application and variables in the meta-language. Meta-abstraction introduces the function space $A \to B$; in contrast, the arrow type of the object language will be noted $A \supset B$.

3.1. Syntax

The original, one-pass call-by-value CPS-transform, reproduced on Figure 1, is presented as a source-to-source program transformation, mapping λ-terms to λ-terms [6]. It therefore consists of two functions: a top-level transformation $[\![\cdot]\!] : M \to M$, and a recursive function with an additional functional argument $[\![\cdot]\!]\cdot : M \to (M \to M) \to M$. If we look at the effect of the transformation on typing, it maps terms M of type A into terms $[\![M]\!]$ of type $[\![A]\!]$, as discussed above. In other words, going through this CPS transform, the syntax is fixed (output syntax = input syntax), but the types vary. There is another way to look at it, where we fix types and let the syntax vary (output syntax $\neq$ input syntax). Let us look at Figure 1 forgetting that the output language is the λ-calculus, and let us find the "tightest" grammar that will satisfy this transformation. Or, in other words, some constructs of the one-pass CPS transform always appear together in output terms, and we can aggregate them into compound syntactic nodes, leading to a specialized syntax of CPS terms. We now detail this analysis.

Let T (respectively S) be the smallest set of terms that are applied to (respectively are the result of applying) a context K. In Equation 4, we return the result of applying the context, so the codomain of function $[\![\cdot]\!]\cdot$ is an element of S. According to Equations 4, 5 and 6, terms in T include variables x and v, and double abstractions, which bodies are the result of the transformation, i.e., a term in S. According to Equations 5, 6 and 7, terms in S include the application of a continuation k to a term in T, a compound of two applications and an abstraction with subterms belonging respectively to T, T and S, and a let-binding, with subterms belonging respectively to T and S. Finally, let P be the codomain of function $[\![\cdot]\!]$. According to Equation 8, it contains an abstraction with subterm in S. Note that we distinguished three kind of variables: *source variables* x, common to the source language, *continuation variables* k and *value variables* v which are generated by the transform. All in all, the specialized target syntax of the one-pass CPS is:

$$
\begin{array}{ll}
S ::= k\ T \mid T\ T\ (\lambda v.\, S) \mid \mathbf{let}\, x = T\, \mathbf{in}\, S & \text{Serious terms} \\
T ::= \lambda x k.\, S \mid x \mid v & \text{Trival terms} \\
P ::= \lambda k.\, S & \text{Programs}
\end{array}
$$

Terms in S, or *serious terms* after Reynolds, represent computations; terms in T, or *trivial terms*, represent values. A *program* P abstracts a serious term over an initial continuation k. The first two constructs of serious terms can be seen respectively as *bind* ($T\ U\ (\lambda v.\, S)$ and *return* ($k\ T$) in the continuation monad. Note however the difference between a bind, which binds a value variable v to the value of the application in the source expression, and a **let**, which binds a source variable x to any trivial term, and stems from a **let** in the source expression. Interestingly, value variables have been shown to occur linearly and non-commutatively, as a stack, and there needs to be only one continuation variable in scope at any point of a CPS term [9].

All in all, the specialized one-pass CPS-transform is defined by the equations of Figure 1, but with the types:

$$
\begin{array}{rcl}
[\![\cdot]\!]\cdot & : & M \to (T \to S) \to S \\
[\![\cdot]\!] & : & M \to P
\end{array}
$$

3.2. Typeful transformation

We chose a type system for the source language—simple types—and we now have a target syntax and a specialized transform that maps to it. Can we assign types to the target syntax so that the transform preserves types? In other words, how should we annotate the syntax above with GADTs so that the following code, a transliteration of the functions on Figure 1, is accepted by OCaml?

```
let rec cps : type a. a m → (a t → s) → s = function
  | MVar x → fun c → c (TVar x)
  | MLam m → fun c → c (TLam (fun x k → cps (m x) (fun v → SRet (k, v))))
  | MApp (m, n) → fun c → cps m (fun t → cps n (fun u → SBind (t, u, fun v → c (TVal v))))
  | MLet (m, f) → fun c → cps m (fun m → SLet (m, fun x → cps (f x) (fun n → c n)))

let cps : α. α m → α p = fun m → PInit (fun k → cps m (fun v → SRet (k, v)))
```

Look at the top-level function: the types assigned to CPS programs are the same as their direct-style image: an α `m` is mapped to an α `p`. It turns out that there is only one possible type assignment satisfying this constraint, the following GADT:

```
type s =                        (* Serious terms *)
  | SRet : α k * α t → s
  | SBind : (α → β) t * α t * (β v → s) → s
  | SLet : α t * (α x → s) → s
and α t =                       (* Trivial terms *)
  | TLam : (α x → β k → s) → (α → β) t
  | TVar : α x → α t
  | TVal : α v → α t
type α p =                      (* Programs *)
  | PInit : (α k → s) → α p
```

Again, think of the type of value variables α `v` and continuation variables α `k` as abstract; they will be defined in the next section. Serious terms do not carry a type like trivial terms α `t` and programs α `p` do; instead, their constructors act as existential types, matching up trivial terms, continuations and values by their types. Here is, for instance, the translation of the applicator expression $\lambda f x.\, f\, x$, in CPS syntax:

```
let ex : α β. ((α → β) → α → β) p = PInit (fun k →
    SRet (k, TLam (fun f k → SRet (k, TLam (fun x k →
        SBind (TVar f, TVar x, fun v → SRet (k, TVal v)))))))
```

Let us now transliterate the transformation itself directly from Figure 1: As witnessed by its type annotation, this main function preserves types: it maps a source expression of type α to a CPS program of the same type α.

3.3. Evaluation

To get a better understanding of the target CPS syntax, let us give its semantics. The result of evaluating a trivial term is a CPS value, i.e., an OCaml function in CPS, returning an answer of abstract type `o`:

```
type o and α vl = VFun : (α vl → (β vl → o) → o) → (α → β) vl
```

For instance, here is the value corresponding to the applicator expression above:

```
let ex : α β. ((α → β) → α → β) vl =
  VFun (fun (VFun f) k → k (VFun (fun x k → f x (fun v → k v))))
```

As usual during evaluation, variables will be substituted by what they stand for; we therefore instantiate their respective namespaces by their semantics.[2]

```
type α x = X of α vl
type α v = V of α vl
type α k = K of (α vl → o)
```

Unsurprisingly, source and value variables stand for values, and continuation variables stand for continuations, i.e., maps from values to answers.

Evaluation consists of one function from trivial terms to values (`eval`) and one from serious terms to answers (`exec`).

```
let rec exec : s → o = function
  | SLet (t, f) → exec (f (X (eval t)))
  | SRet (K c, t) → c (eval t)
  | SBind (t, u, s) → match eval t with
    | VFun f → f (eval u) (fun v → exec (s (V v)))
and eval : type a. a t → a vl = function
  | TVar (X x) | TVal (V x) → x
  | TLam f → VFun (fun x k → exec (f (X x) (K k)))
```

A let triggers the execution of its body, binding its variable to the value of its definition. A return $k\ T$ evaluates its term T and passes its value to the continuation that k stands for. To execute a bind $T\ U\ (\lambda v.\, S)$, one binds the application of the value of U to the value of T as a new value variable `v`, and continue executing S. The type of `exec` might come as a surprise: since answer type `o` is abstract, executing a serious term has no result (if not for potential side effects if we had included e.g. a `printf` primitive) since we did not specify an initial continuation, i.e. what to do with the result of the computation. In Kennedy's words, "the only observations we can make on [pure CPS] programs is termination" [13]. When evaluating a program, one way to circumvent this is to use exceptions (or assignment [14]) to let the final value jump out of a "trick" initial continuation, catch it at top-level, and return it:

```
let eval : type a. a p → a vl = fun (PInit s) →
  let module M = struct exception E of a vl end in
  try ignore (exec (s (K (fun x → raise (M.E x))))); assert false
  with M.E x → x
```

The second line defines a local exception which will not escape the scope of the function body: if the program was indeed generated by CPS-transformation of a source expression M, its use of continuations is linear [9], which means that the initial continuation will eventually be called once, and the exception triggered. Caught by the surrounding handler, the final value will finally be returned.

4. CPS-transformation of β-redexes

The transformation of Section 3 translates expression $((\lambda xy.\, x)\ a)\ b$, which has two nested β-redexes, into the CPS term:

$$\lambda k.\, (\lambda xk.\, k\ (\lambda yk.\, k\ x))\ a\ (\lambda v.\, v\ b\ (\lambda w.\, k\ w))$$

As you can see, each original β-reduction step takes two substitutions in this CPS form: one for the value and one for the argument's continuation. Yet, continuations are only threaded here, which is

[2] One could object that this instantiation of the domain of variables takes us away from weak HOAS, and make our framework inconsistent. Indeed, one could forge a well-typed term containing a diverging value. This instantiation is only necessary to implement evaluation. It is a commodity to avoid more verbose solutions like De Bruijn indices or parametricity in α `x`.

characteristic of nested β-redexes. This problem has been studied under the name "compacting CPS terms" [17, 8]. Indeed, a more compact CPS-term could however be obtained if we were to see redexes as **let**s in the source expression. The equivalent expression $(\mathbf{let}\, x = a\, \mathbf{in}\, \lambda y.\, x)\ b$ is translated into:

$$\lambda k.\, \mathbf{let}\, x = a\, \mathbf{in}\, (\lambda y.\, x)\ b\ (\lambda v.\, k v)$$

which exposes the second redex; extra mileage is therefore obtained by seeing all *nested* β-redexes as **let**s. The equivalent expression $\mathbf{let}\, x = a\, \mathbf{in}\, \mathbf{let}\, y = b\, \mathbf{in}\, x$ is translated into:

$$\lambda k.\, \mathbf{let}\, x = a\, \mathbf{in}\, \mathbf{let}\, y = b\, \mathbf{in}\, k\ x$$

This residual CPS term is more compact and efficient; syntactically, it has no apparent β-redexes anymore. The formalization and typing of the "compacting CPS" optimization proposed by Danvy and Nielsen [8] is the object of this section.

4.1. Syntax

Now, we want to write all explicit redexes in the target language as lets. Let us alter the syntax of CPS terms of Section 3 so as to ban the construction of redexes, such as $(\lambda x k.\, S_1)\ T\ (\lambda v.\, S_2)$. In other words, the functional term T_1 in a "bind" $T_1\ T_2\ (\lambda v.\, S)$ must only be an *identifier*, i.e., a source or a value variable. We thus stratify the syntax of trivial terms into identifiers I and functions, and constrain the functional subterm of a "bind" to be an identifier, which rules out the possibility to write redexes:

$$\begin{array}{rr}
P ::= \lambda k.\, S & \text{Programs} \\
S ::= k\ T \mid I\ T\ (\lambda v.\, S) \mid \mathbf{let}\, x = T\, \mathbf{in}\, S & \text{Serious terms} \\
T ::= \lambda x k.\, S \mid I & \text{Trival terms} \\
I ::= x \mid v & \text{Identifiers}
\end{array}$$

This stratification is akin to the traditional syntax of normal forms of the (direct-style) λ-calculus, which stratify the syntax into normal forms and atomic terms. Identifiers play the role of atomic terms, trivial terms of normal forms; the applications nodes, which in the syntax of normal forms belong to atomic terms, are here "extruded" to binds in serious terms, which value variables range over.

4.2. Typeful transformation

Encoding the typed syntax is trivial; we just adapt the previous typing rules to the new, constrained syntax:[3]

```
type α i = IVar of α x | IVal of α v (* Identifiers *)
type s = (* Serious terms *)
  | SBind : (α → β) i * α t * (β v → s) → s
  | SRet : α k * α t → s
  | SLet : α t * (α x → s) → s
and α t = (* Trivial terms *)
  | TLam : (α x → β k → s) → (α → β) t
  | TIdent : α i → α t
type α p = PInit of (α k → s) (* Programs *)
```

[3] Incidentally, if you see this typed language as a proof system through the Curry-Howard isomorphism, you will notice striking similarities with the LKQ focused sequent calculus. It is more than a coincidence, and is the subject of a work in preparation.

Now comes the transformation itself. Danvy and Nielsen's insight was that for the transformation to be done in one pass, it needs to be *context-sensitive*: the result of transforming the innermost beta-redex $(\lambda xy.\, x)\, a$ of the introductory example should "remember" the outermost application node, which forms a nested beta-redex. Each nested beta-redex is transforms like a let, and not like a regular application. Technically, their transformation was annotated by an integer, encoding how many nested redexes are in the context at any point; the higher-order type of the translation function depended on this integer.

Despite the absence of dependent types in OCaml, we can encode, as a GADT, the relation between the type of the input context and the type of the output value:

```
type (α, ξ) l =
  | Base : (α, α t) l
  | Arr : (β, ξ) l → (α → β, α t → (ξ → s) → s) l
```

Each inhabitant of type $(\alpha,\ \xi)$ `l` is a proof that an expression α `m` will be translated into a value ξ. For instance:

```
let ex : α β γ. (α → β → γ, α t → ((β t → (γ t → s) → s) → s) → s) l =
  Arr (Arr Base)
```

The main translation function follows:

```
let rec cps : type a x. (a, x) l → a m → (x → s) → s = fun l → function
  | MVar x → fun c → c (psi l (IVar x))
  | MApp (m, n) → fun c → cps (Arr l) m (fun m → cps Base n (fun n → m n (fun a → c a)))
  | MLet (m, n) → fun c → cps Base m (fun t → SLet (t, fun x → cps l (n x) (fun u → c u)))
  | MLam m → fun c → match l with
    | Base → c (TLam (fun x k → cps Base (m x) (fun m → SRet (k, m))))
    | Arr l → c (fun t k → SLet (t, fun x → cps l (m x) k))
```

It differs from its ancestor of Section 3 in the following points. First, it awaits one more argument, which encodes the relation above between the type of the input context `a` and the type of the output `x`; the type of the function is then explicit: it maps an input term of type `a` to a value of type (`x` → `s`) → `s`. The application case MApp does not build an SBind node right away as in the previous section: instead, it uses OCaml's application to combine the results of the translations of its subterms; eventually, the SBind node are produced at the variables; the auxiliary function `psi` unrolls the type relation and generates the function building the binds:

```
let rec psi : type a x. (a, x) l → a i → x = function
  | Base → fun i → TIdent i
  | Arr l → fun i t c → SBind (i, t, fun v → c (psi l (IVal v)))
```

At top level, we start in the empty context, obviously:

```
let cps : α. α m → α p = fun m → PInit (fun k → cps Base m (fun x → SRet (k, x)))
```

Once again, the type annotation of this top-level function says it all: it guarantees that direct-style terms are mapped to CPS programs in our restricted syntax (with no apparent redexes), and that types are preserved.

5. CPS-transformation of tail calls

The transformation of Section 4 translates expression $\lambda x.\, f\, x\, (g\, x)$ into the CPS term:

$$\lambda x k.\, g\ x\ (\lambda v_1.\, f\ x\ (\lambda v_2.\, v_2\, v_1\, (\lambda v_3.\, k\, v_3)))$$

Each of the source application nodes are translated into binds, each carrying a *continuation* $(\lambda v_i. \ldots)$. The second continuation, $(\lambda v_3.\, k\ v_3)$ is however trivial: it immediately returns the computed value to the initial continuation, which is characteristic of tail calls. Indeed, when executing this term (Section 3.3), the last continuation will allocate one stack frame, to then immediately pass control to the initial continuation. In fact, if we read for a minute this CPS term as a traditional λ-term, this continuation is an η-redex. For efficiency and conciseness, it is necessary to avoid the detour and directly translate tail call continuations into continuation variables: $\lambda x k.\, g\ x\ (\lambda v.\, f\ v\ k)$. One solution would be to detect tail calls *a posteriori*, after CPS translation, and mark them as such; but it would require more computation and memory, so as before we embark in solving the problem in one pass. The formalization of this optimization, inspired by Danvy and Filinski's solution [6] but combined with the previous β-redexes optimization, is the object of this section.

5.1. Syntax

Let us alter the syntax of CPS terms of Section 4 so as to accept the contracted serious term $f\,v\,k$, and reject its expanded form $f\,v\,(\lambda w.\, k\,w)$. This is delicate because obviously, the first "bind" $g\,x\,(\lambda v.\, f\,v\,k)$ must still be legal, so we cannot ban the production $T\,U\,(\lambda v.\, S)$ altogether from our grammar. In essence, we want to ban continuations that are trivial, i.e., which amount to a return, and replace them simply by the continuation variable.

For this, we introduce the grammar entry of (non-trivial) *computations* U, distinguishing them from any serious term S (trivial or not). Serious terms can be actual computations U or trivial ones $(k\,T)$ but computations cannot be trivial. Now, a *continuation* C, i.e., the last constituent of a bind, can then either refer directly to a continuation variable (the contracted form) or bind a value variable (the expanded form). In the latter case, the continuation cannot be a return: it needs to be a computation U. All in all, the syntax of properly tail-recursive CPS terms is:

$P ::= \lambda k.\, S$	Programs
$S ::= k\ T \mid U$	Serious terms
$U ::= I\ T\ C \mid \mathbf{let}\, x = T\, \mathbf{in}\, S$	Computations
$C ::= \lambda v.\, U \mid k$	Continuations
$T ::= \lambda x k.\, S \mid I$	Trival terms
$I ::= x \mid v$	Identifiers

Note that a **let** is a computation U, even if its body can be any term S. Note also that this stratification rules out the possibility to write $I\ T\ (\lambda v.\, k\ T')$, whatever T' is. Yet, as the next section will show, this set of terms is large enough to be the image of CPS.

5.2. Typeful transformation

The modified syntax is easily typed and encoded into a GADT:

```
type s = SRet : α k * α t → s | SComp : u → s
and u = UBind : (α → β) i * α t * β c → u | ULet : α t * (α x → s) → u
and α c = CCont : (α v → u) → α c | CTail : α k → α c
and α t = TIdent : α i → α t | TLam : (α x → β k → s) → (α → β) t
type α p = PInit of (α k → s)
```

The new constructor is `CTail k`, representing a tail continuation k.

Now, the transformation relies on the observation that being in tail position in the λ-calculus is an inherited attribute: whether a function call is a tail call can be determined only by the context in which it appears, i.e., we can stratify its syntax into two syntactic categories:

$$M ::= \lambda x.\, M' \mid x \mid \mathbf{let}\, x = M\, \mathbf{in}\, M \mid M\ M \qquad \text{Terms in any position}$$
$$M' ::= \lambda x.\, M' \mid x \mid \mathbf{let}\, x = M\, \mathbf{in}\, M' \mid M\ M \qquad \text{Terms in tail position}$$

We choose to track this information during the CPS transformation itself, by splitting the previous CPS function into two mutually recursive ones, depending on the position of the sub-term translated. In tail position, we will use `cpst`; in another position, we will use `cps`:

```
(* all this is as before (except that it returns a u *)
let rec cps : type a x. (a, x) l → a m → (x → u) → u = fun l → function
  | MVar x → fun c → c (psi l (IVar x))
  | MApp (m, n) → fun c → cps (Arr l) m (fun m → cps Base n (fun n → m n (fun a → c a)))
  | MLet (m, n) → fun c → cps Base m (fun t →
      ULet (t, fun x → SComp (cps l (n x) (fun u → c u))))
  | MLam m → fun c → match l with
    (* changes start here: *)
    | Base → c (TLam (fun x k → cpst (m x) k)) (* subterm is tail *)
    | Arr l → c (fun t k → ULet (t, fun x → SComp (cps l (m x) k)))
    (* new: the lambda is part of a nested redex in tail position so we produce a let with a body
    in tail position *)
    | Tail → c (fun t k → ULet (t, fun x → cpst (m x) k))

(* identical to the previous section, except... *)
and cpst : type a. a m → a k → s = function
  | MVar x → fun k → SRet (k, TIdent (IVar x))
  | MLam m → fun k → SRet (k, TLam (fun x k → cpst (m x) k))
  | MLet (m, f) → fun k → SComp (cps Base m (fun m → (* m is not in tail position *)
      ULet (m, fun x → cpst (f x) k)))
  | MApp (m, n) → fun k → SComp (cps Tail m (fun m → (* m is a function in tail position *)
      cps Base n (fun n → m n k)))
```

First, look at the types of these functions: the translation of a term in tail position `cpst` can give rise to any serious term S, computation or otherwise; but the translation of a term in non-tail position will have to return a computation U. Function `cps` takes a "translation-time" continuation function `x → u` as before (and therefore also a witness `l` of the type relation). Function `cpst` does not need such arguments: a tail call cannot have a context of nested redexes above him; therefore it only takes a "run-time" continuation α `k` that will eventually be inserted, marking the tail call.

Notice the new case when transforming a non-tail abstraction: we can now detect "at a distance" the lambdas that will be eventually give rise to redexes in tail positions (constructor `Tail`). This case is generated when transforming an application in tail position.

This new constructor is added to the GADT `l` that relates source and target types:

```
type (α, ξ) l =
  | Base : (α, α t) l
  (* modification here: changed s into u *)
  | Arr : (β, ξ) l → (α → β, α t → (ξ → u) → u) l
  (* new: tail position indicator. This marks a term in functional position of a tail
     application. *)
  | Tail : (α → β, α t → β k → u) l
```

There is now two ways to relate function types: those in tail position, which therefore do not need a functional argument, and those who are not in tail position, as before. The function `psi` generating the binds is modified accordingly: it generates proper tail calls CTail when indicated by the Tail relation.

```
(* same as before *)
let rec psi : type a x. (a, x) l → a i → x = function
  | Base → fun i → TIdent i
  | Arr l → fun i t c → UBind (i, t, CCont (fun v → c (psi l (IVal v))))
  (* new case: where we generate a tail bind *)
  | Tail → fun i t k → UBind (i, t, CTail k)
```

Finally, the top-level function calls the translation in tail position:

```
let cps : α. α m → α p = fun m → PInit (fun k → cpst m k)
```

Its type annotation guarantees that all source terms are translated into CPS programs, with their types preserved.

5.3. Evaluation

To showcase the gain of this optimization, let us to conclude with the evaluation function for the modified syntax:

```
let rec exec : s → o = function
  | SRet (K c, t) → c (eval t)
  | SComp c → compute c
and compute : u → o = function
  | ULet (t, f) → exec (f (X (eval t)))
  | UBind ((IVar (X (VFun f)) | IVal (V (VFun f))), t, c) → f (eval t) (continue c)
and continue : type a. a c → a vl → o = function
  | CCont s → fun v → compute (s (V v))
  | CTail (K c) → c
and eval : type a. a t → a vl = function
  | TIdent (IVar (X x) | IVal (V x)) → x
  | TLam f → VFun (fun x k → exec (f (X x) (K k)))
```

As you can see, the new function `continue` has two cases: in one (non-tail calls), a closure is allocated; in the other (tail calls), the continuation `c` is immediately returned. Tail calls are therefore more efficient at run time.

6. Related and further work

Optimizing CPS translations By exploring the design space of CPS translations, we hope to have given a better understanding of Plotkin's original translation, and paved the way to implementing safe and optimizing compilers based on call-by-value CPS, using only a general-purpose programming language and no external proof tools. Of course, there is still much work to scale this study to a real-world functional programming language, notably by adding syntactic constructs (abstract data types and pattern-matching[4], exceptions and other control operators...), and optimizations (CSE or some other partial evaluation mechanism). This work is by no means the first formalization of a CPS

[4]We have already proposed such an encoding of pattern-matching at `syntaxexclamation.wordpress.com/2014/04/12/representing-pattern-matching-with-gadts/`, which could be a starting point.

transform (see [10] for a survey); it relies heavily on previous study of CPS and CPS optimizations [6, 8, 2].

Beyond compiler design, we are currently looking at the logical interpretation of these optimizations through the Curry-Howard isomorphism. Our source language, the simply-typed λ-calculus, corresponds to Natural deduction; what logical system are in correspondence with the different intermediate languages we have designed here? Striking connections to the proof-theoretic notion of *focusing* [1] have already been observed.

Typeful programming We have called "typeful" our style of lightweight formalization: infer dedicated syntax and constrain it so that it acts as a specification, prove type preservation by annotating the transformations with simple types, encoded by GADTs in OCaml. Besides the obvious interest of the formal guarantee, it also gives us a methodology to discover novel properties of the transformations (like the typing rules governing the dedicated syntax in Section 3). This research program was initiated with the formalization of Normalization by Evaluation [7] which gave us insight on the nature of normalization in the presence of sums. An endeavor sharing the same name was carried out by Chen et. al. [4], but in a slightly different context: proving type preservation in a dependently-typed setting (DML) with De Bruijn indices, with no particular focus on the output syntax nor on optimizations.

One question arises naturally: can we extend this encoding beyond simple types, adding e.g., primitive types, polymorphism etc. It is to our best knowledge an open issue whether polymorphism à la System F [12] can be deeply embedded into OCaml's type system. Finally, we have looked at a call-by-value strategy here, other strategies like call by name or call by need will surely be of interest as well, and offer insights on their behavior.

Acknowledgements

This work was carried out mainly when the author was employed at Aarhus University. We would like to thank Olivier Danvy who motivated and guided it, and transmitted us his passion for the topic.

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992. 13

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 2006. 1, 4, 13

[3] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009. 3

[4] C. Chen, R. Shi, and H. Xi. Implementing typeful program transformations. *Fundam. Inform.*, 69(1-2):103–121, 2006. 13

[5] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. 2, 3

[6] O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. 1, 4, 5, 10, 13

[7] O. Danvy, C. Keller, and M. Puech. Typeful normalization by evaluation. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12-15, 2014, Paris, France*, volume 39 of *LIPIcs*, pages 72–88. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. 2, 13

[8] O. Danvy and L. R. Nielsen. CPS transformation of beta-redexes. *Inf. Process. Lett.*, 94(5):217–224, 2005. 1, 8, 13

[9] O. Danvy and F. Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Feb. 1995. 5, 7

[10] Z. Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels. (Formal verification of an optimizing compiler for functional languages)*. PhD thesis, Paris Diderot University, France, 2009. 1, 13

[11] J. Garrigue and D. Rémy. Ambivalent types for principal type inference with gadts. In C. Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013. 2, 3

[12] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. 3, 13

[13] A. Kennedy. Compiling with continuations, continued. In R. Hinze and N. Ramsey, editors, *ICFP*, pages 177–190. ACM, 2007. 7

[14] O. Kiselyov. Monads for undelimited continuations, 2011. Web post, available at `okmij.org/ftp/continuations/undelimited.html#proper-contM`. 7

[15] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. 1

[16] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. 1, 2, 4

[17] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993. 8

[18] M. Wand. Correctness of procedure representations in higher-order assembly language. In S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA, March 25-28, 1991, Proceedings*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311. Springer, 1991. 4

[19] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008. 3

[20] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In A. Aiken and G. Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. 2