INFORMATICA, 1991, Vol.2, No.4, 524-538

CONTINUATION TRANSFORMATION AND REDUCTION

Kęstutis URBAITIS

Institute of Mathematics and Informatics,
Lithuanian Academy of Sciences,
2600 Vilnius, Akademijos St.4, Lithuania

Abstract. The continuation transformation, its application for recursion removal and relation with reduction algorithms is analyzed. The reduction algorithm for expressions in continuation form is presented.

Key words: continuations, control, reduction, recursion.

1. Introduction. Continuation is a device which is used in formalizing the notion of control flow in programming languages and their semantics. It represents the formal notion of a current point in a computation. Contrary to the lower level notions of a return address and return address stack, continuations capture control environment in full and use higher-order functions to achieve this.

Originally continuations where invented in denotational semantics to describe control structures, such as *goto* operators, exits, nonlocal jumps, label values (Strachey and Wadsworth, 1974). The metalanguage of denotational descriptions is purely functional, based on the λ-calculus. Denotational equations are usually written in continuation-passing style and map various nonfunctional constructions of the object language into pure λ-calculus.

In case of imperative languages, where there is an explicit sequence of operators, a continuation in a certain point of a program represents the rest of the operators from that point, that is the remaining part of the program.

In applicative expression languages programs have a tree structure. Continuations represent a certain point in a tree by splitting it into a subexpression and the rest of the tree above it, or *context.* A continuation is a special representation of a context.

Continuations can be used not only in metalanguage, but on the object level as well. Any function can be translated into continuation-passing form by using continuation transformation. Every continuation-passing function receives an additional continuation parameter. Instead of simply returning, the function either passes its result directly to this continuation, or passes this continuation, possibly nested into a new larger expression, to some other function.

Functions in continuation form get explicit access to their dynamic context. They may use this access in different ways: instead of returning to dynamic context, as usual, they can throw it away, thus implementing escape, or change it, thus performing nonlocal jump. By manipulating continuations it is possible to implement complex sequential control structures: recursion, backtracking, nonlocal jumps, escapes, interrupts, coroutines. This possibility to process continuations explicitly allows to translate languages, extended with non-functional control constructs into purely functional continuation-passing programs.

An algorithm based on continuation transformation can be used for recursion removal in functional programs: it transforms any recursive program into continuation form, which is tail-recursive, i.e. iterative. The transformation does not change the algorithm of the program and does not reduce its complexity, only makes recursive control explicit.

We shall be interested in continuation transformation for purely functional expression languages, based on λ-calculus.

λ-expressions are built from variables by application and λ-abstraction operations:

variable: v is λ-expression, if v – variable,

application: MN is λ-expression, if M, N – λ-expressions,

abstraction: $\lambda v.M$ is λ-expression, if v – variable, M – λ-expression.

Expressions have a structure of trees, complex expressions contain subexpressions. Every subexpression has a context, which is the part of the expression surrounding it. Context can be defined as an expression with a hole in it and will be denoted $C[\ ]$. When the hole of context $C[\ ]$ is filled with subexpression N it will be written $C[N]$.

Continuation transformation can be generalized for λ-calculus. Contexts can be interpreted as functions with one parameter, where the parameter replaces the hole in the context:

$$C[\ ] \longrightarrow \lambda v.C[v].$$

The context controls how the computation will continue in the absence of explicit control transfers. It expects to receive the value returned by the subexpression, with which it will continue the computation.

A continuation of a λ-expression is this special functional encoding for its context. Informally, a continuation is an additional argument of a function and continuation transformation replaces the context of a subexpression by its functional encoding which is passed to the subexpression as an additional argument:

$$C[M] \longrightarrow M\lambda v.C[v],$$

where the original expression can be reconstructed by applying the continuation of a subexpression to the subexpression itself.

In general case, the expression to be transformed already

has initial continuation k, and transformation rule then becomes:

$$C[M]\ k \longrightarrow M\ (\lambda v.k\ C[v]) \qquad (TrI)$$

This transformation step must be applied recursively to remaining subexpressions. If we denote the transformed image of M as $\overline{M}$, the transformation rule can be written as equation:

$$\overline{C[M]}\ k = \overline{M}\ (\lambda v.\ \overline{C[v]}\ k) \qquad (TrP)$$

Computation is modelled in λ-calculus by *reduction*. Reduction relations may be defined in several stages. The basic relation is the rule of β-reduction:

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N] \qquad (\beta)$$

where the left side is a *β-redex* and the right side means substituting N for variable x in M, renaming other variables, if necessary, to avoid variable name capture.

Next the one step β-reduction relation $\longrightarrow_\beta$ is defined as the compatible closure of $\xrightarrow{\beta}$:

$$M \xrightarrow{\beta} N \Longrightarrow M \longrightarrow_\beta N$$
$$M \longrightarrow_\beta N \Longrightarrow ZM \longrightarrow_\beta ZN \qquad (Cr)$$
$$M \longrightarrow_\beta N \Longrightarrow MZ \longrightarrow_\beta NZ \qquad (Cl)$$
$$M \longrightarrow_\beta N \Longrightarrow \lambda x.M \longrightarrow_\beta \lambda x.N \qquad (\xi)$$

Usually, in implementations of functional programming languages, the rule (ξ) is omitted. This rule would require to reduce bodies of defined functions even before they are called, which is unnatural in practical applications. β-reduction without the (ξ) rule is called *weak reduction*.

Finally, the β-reduction relation $\twoheadrightarrow_\beta$ is defined as the reflexive, transitive closure of $\longrightarrow_\beta$:

$$M \twoheadrightarrow_\beta M \qquad (Ref)$$
$$M \longrightarrow_\beta Z,\ Z \twoheadrightarrow_\beta N \Longrightarrow M \twoheadrightarrow_\beta N \qquad (Tran)$$

Different strategies may be used to reduce λ-expressions. The two most common and simple are innermost and outermost. In programming, the innermost strategy is called 'by-value', the outermost – 'by-name'. Both strategies have

special meaning when weak reduction is used. We shall further suppose the use of the innermost weak reduction strategy. λ-abstractions and variables will be called *values*.

2. Continuation transformation for recursion removal. We shall introduce continuation transformation by examples of recursion removal (Wand, 1978; Wand, 1980).

Continuation transformation can be used to transform recursive functions into tail-recursive, or iterative functions. The idea is to split function body into context and subexpression so that recursive call would be on the top of the subexpression. The context is transformed into a continuation which is passed to the function as an additional argument in the recursive call. As a result of this transformation the recursive call is lifted from the inside of the function body to the top level and function definition becomes tail-recursive, or iterative.

The transformation can be demonstrated by the following example of list reversal:

Example 1. The naive list reverse function:

$$rev\ [\,] = [\,]$$
$$rev\ [m : n] = append\ (rev\ n)[m],$$

where $[m : n]$ denotes a list with the head m and tail n, *append* is list concatenation function. We define functions by equations with pattern matching, which can be easily translated into λ-calculus. We shall use $-, +$ and $*$ as the usual arithmetic infix operators. The same function in continuation form gets an additional continuation parameter k:

$$rev'\ [\,] \qquad k = k[\,]$$
$$rev'\ [m : n]\ k = rev'\ n(\lambda v.\ k\ (append\ v[m])).$$

By induction the following property can be proved:

$$rev'\ n\ k = k\ (rev\ n).$$

So we can define the original function by providing the transformed function with the initial continuation-identity function I:

$$rev\ n = rev'\ n\ I.$$

Programs in continuation form can be further transformed and optimized. One of the methods of optimization is to replace continuations by data structures. In the previous example continuation can be replaced by a list, and the expensive call to *append* can be deleted:

$$rev''\ [\,]\quad a = a$$
$$rev''\ [m : n]a = rev''\ n\ [m : a].$$

We can prove by induction:

$$rev''\ x\ a = append\ (rev\ x)\ a$$
$$rev''\ x\ [\,] = rev\ x.$$

The transformed functions have the property that they are tail-recursive, or iterative. They have explicit access to their context through their continuations, which they invoke instead of the usual return to the context from recursive call in the original functions.

The factorial function is transformed similarly:

Example 2. Factorial:

$$fac\ 0 = 1$$
$$fac\ n = n * (fac\ (n-1)).$$

After transformation:

$$fac'\ 0 = \lambda k.k\ 1$$
$$fac'\ n = \lambda k.fac'\ (n-1)\ (\lambda v.k\ (v * n)).$$

The function calculating Fibonacci numbers has double recursion and requires to apply the transformation twice:

Example 3. Fibonacci numbers:

$$fib\ 0 = 1$$
$$fib\ 1 = 1$$
$$fib\ n = fib\ (n-1) + fib\ (n-2).$$

After transformation:

$$fib'\ 0 = \lambda k.k\ 1$$
$$fib'\ 1 = \lambda k.k\ 1$$
$$fib'\ n = \lambda k.fib'\ (n-1)(\lambda v_1.fib'\ (n-2)(\lambda v_2.k\ (v_1 + v_2))).$$

Next we shall reformulate continuation transformation in *equational* style, by eliminating λ-abstractions.

It is possible to write continuation functions without using λ-abstractions in a purely applicative equational notation by using local definitions. The factorial function can be rewritten in the following way:

$$fac\ 0\ k = k\ 1$$
$$fac\ n\ k = fac\ (n-1)\ k_1,$$

where $k_1\ v = k\ (v * n)$.

Local definitions can be made global by lifting their nonlocal variables. The following replacement must be made in factorial:

$$k_1 \longrightarrow c_1\ n\ k.$$

This technique is known in functional programming as $\lambda - lifting$ or *supercombinator abstraction* algorithm. For factorial function we get:

$$fac\ 0\ k = k\ 1$$
$$fac\ n\ k = fac\ (n-1)\ (c_1\ n\ k)$$
$$c_1\ n\ k\ v = k\ (v * n).$$

Applying the same transformations to Fibonacci function gives:

$$fib\ 0\ k = k\ 1$$
$$fib\ 1\ k = k\ 1$$
$$fib\ n\ k = fib\ (n-1)\ (c_1\ n\ k)$$
$$c_1\ n\ k\ v_1 = fib\ (n-2)\ (c_2\ n\ k\ v_1)$$
$$c_2\ n\ k\ v_1\ v_2 = k\ (v_1 + v_2).$$

It is possible to define continuation transformation in equational form directly, instead of lifting transformed λ-expressions.

Let the function definition be given by the following equation:

$$f\ x\ y\ \ldots\ z = T,$$

where T – applicative expression.

Let T have the form $T \equiv C[M]$. Continuation transformation produces from this equation the following equation by introducing continuation function c:

$$f'\ x\ y\ \ldots\ z\ k = M\ (c\ x\ y\ \ldots\ z\ k)$$

and additional equation is generated, defining continuation function c:

$$c\ x\ y\ \ldots\ z\ k\ v = k\ C[v].$$

This transformation exactly corresponds to rule (TrI), introduced earlier.

When transformation is used for recursion removal, factorization is performed as earlier described for subexpressions starting with recursive calls. Transformation step is repeated for all equations, including the new ones, until all recursion is removed.

3. General continuation transformation. Continuation transformation can be generalized for λ-expressions by applying factorization to all applications. λ-expressions are transformed into continuation form by the following algorithm (Meyer and Wand, 1978):

Algorithm 1. Continuation transformation:

$$\overline{x} = \lambda k.k\ x \qquad (Var)$$

$$\overline{\lambda x.M} = \lambda k.k\ (\lambda x.\overline{M}) \qquad (Abs)$$

$$\overline{M\ N} = \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.m\ n\ k)) \qquad (App)$$

The additional λ-abstractions which are introduced on the right are continuations and variable k is the initial continuation for the transformed expression. Because every $\overline{M}$ is an abstraction, the transformation introduces new β-redexes. By reducing these redexes we get a simplified representation of transformed expression. As a result of these reductions the variables and λ-abstractions which are initially factored out get back into their original positions and the continuations of applications are concatenated. In future we shall assume this simplification implicitly. It is easy to notice that continuation transformation is a generalization of rule (TrP).

The main property of the transformation is that it removes nested function applications by factoring them out. The absence of nested applications is preserved under β-reduction.

There is at most one redex which is not inside the scope of a λ-abstraction. Thus by-value and by-name weak reductions coincide.

It is possible to develop the algorithm which produces the simplified transformation directly by factoring out the nested applications only.

Algorithm 2. Continuation transformation by rewriting:

It starts by introducing the initial continuation k:

$$M\ N \longrightarrow \lambda k.M\ N\ k \qquad (Ini)$$

Next, the following rewrite rules are applied to such redexes that k contains k until possible:

$$\lambda k.(M\ L)\ N\ K \longrightarrow \lambda k.M\ L\ (\lambda v.v\ N\ K) \qquad (Fl)$$

$$\lambda k.M\ (N\ L)\ K \longrightarrow \lambda k.N\ L\ (\lambda v.M\ v\ K) \qquad (Fr)$$

where v is a new variable. The algorithm is applied recursively to the bodies of λ-expressions.

The recursion removal algorithm described in the previous section can be generalized for λ-expressions in the following way:

Algorithm 3. Equivalent continuation transformation:

$$\overline{\overline{x}} = \lambda k.k\ x \qquad (Var)$$

$$\overline{\overline{\lambda x.M}} = \lambda k.k\ (\lambda x.\overline{\overline{M}}) \qquad (Abs)$$

$$\overline{\overline{M\ N}} = \lambda k.\overline{\overline{M}}\ (\lambda m.\overline{\overline{N}}\ (\lambda n.k\ (m\ n))) \qquad (App)$$

It can be proved by β-conversion and induction:

$$\overline{\overline{M}} = \lambda k.k\ M$$

$$\overline{\overline{M}}\ I = M.$$

Algorithm 3 corresponds to rule (TrI).

Algorithms 1 and 3 differ only in the third rule (App) and in general produce nonequivalent expressions:

$$\overline{\overline{M}} \neq \overline{M}$$

$$\overline{M} \neq \lambda k.k\ M.$$

The aim of the next section is to construct the reduction algorithm for expressions with continuations $\overline{M}$, which produces the same result as the ordinary β-reduction algorithm for M.

4. Reduction. We shall investigate the algorithms of λ-expression reduction. The usual by-value or leftmost-innermost algorithm of weak reduction will be used (Felleisen and Friedman, 1986), that is, function application will be reduced by first reducing its function part, next reducing its argument part and then proceeding to reduce the body of the λ-abstraction, after performing the substitution. Variables and λ-expressions reduce to themselves.

The semantics of λ-expressions is usually denotationally defined by means of a function $[\![\ ]\!]$, which takes additional parameter-environment. Environment is a function mapping variable names to their values.

$$\begin{aligned}
&[\![x]\!] \quad e = e\ x\\
&[\![\lambda x.M]\!]\ e = \lambda v.[\![M]\!]\ e[x := v]\\
&[\![M\ N]\!]\ e = [\![M]\!]\ e\ ([\![N]\!]\ e).
\end{aligned}$$

This description is called direct semantics. It can be transformed into continuation, or indirect semantics by applying continuation transformation to function $[\![\ ]\!]$:

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

We shall not be interested in environment manipulation, but only in expression traversal, so we may omit environment parameters:

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

This semantic function is very similar to continuation transformation of λ-expressions. After replacing λ-abstractions with local definitions, the rule for application can be rewritten:

$$[\![M\ N]\!]\ k = [\![M]\!]\ k_1,$$

where $k_1\ m = [\![N]\!]\ k_2$, $k_2\ n = m\ n\ k$.

The local definitions of continuation functions k_1 and k_2

can be made global by λ-lifting their nonlocal variables, making following replacements:

$$k_1 \longrightarrow k_1\ N\ k$$
$$k_2 \longrightarrow k_2\ m\ k.$$

With these changes the rule for application can be replaced by three rules:

$$[\![M\ N]\!]\ k = [\![M]\!]\ (k_1\ N\ k) \qquad (App)$$
$$k_1\ N\ k\ m = [\![N]\!]\ (k_2\ m\ k) \qquad (Swap)$$
$$k_2\ m\ k\ n = m\ n\ k \qquad (Beta)$$

Next we shall analyze the weak by-value reduction algorithms of λ-expressions.

Algorithm 4. Reduction of λ-expressions (recursive):

$$x \longrightarrow x. \qquad (Var)$$
$$\lambda x.M \longrightarrow \lambda x.M. \qquad (Abs)$$
$$M\ N \longrightarrow T[x := R] \Longleftarrow M \longrightarrow \lambda x.T,\ N \longrightarrow R. \qquad (App)$$

This algorithm is recursive and corresponds to direct semantics. Our aim is to construct a non-recursive reduction machine, the transitions of which constitute a tail-recursive rewriting system. Contrary to reductions, transitions are always applied to the whole expression. Reductions can be performed on subexpressions, because the one-step reduction relation $\longrightarrow_\beta$ is defined as a compatible closure of $\longrightarrow_\beta$. To eliminate recursion the continuation transformation can be applied to *Algorithm 4*. The resulting algorithm corresponds to the function of continuation semantics.

The classical SECD machine implements recursion by means of a stack. The SECD reduction machine uses environment register E for accumulating delayed substitutions and stack register S for context saving during expression traversal. We present one version of the SECD machine without environment. It contains two registers: C – *Control* – for current expression, and S – *Stack*. Stack grows to the left, its elements are separated by dots '.' .

Algorithm 5. SECD reduction machine:

Control	Stack		Control	Stack	
x	S	$\longrightarrow$	$-$	$x.S$	(Var)
$(\lambda x.M)$	S	$\longrightarrow$	$-$	$(\lambda x.M).S$	(Abs)
$(M\ N)$	S	$\longrightarrow$	M	$(\_\ N).S$	(App)
$-$	$m.(\_\ N).S$	$\longrightarrow$	N	$(m\ \_).S$	$(Swap)$
$-$	$n.(\lambda x.m\ \_).S$	$\longrightarrow$	$m[x := n]$	S	$(Beta)$

Variables and λ-abstractions are loaded on stack unreduced (Var and Abs).

For application $M\ N$ both operands, function M and argument N, are first reduced and then substitution performed. There are three rules for application: for reduction of M (App), reduction of N ($Swap$) and substitution ($Beta$). When application $M\ N$ is encountered, machine transfers to reduce M. Argument N is loaded on stack in the form $(\_\ N)$, which shows that N is an unreduced argument. When M is reduced, its value m is loaded on stack by (Var) or (Abs) and then swapped with N by ($Swap$). m is saved on stack in the form $(m\ \_)$, which shows that m is a reduced function. Finally, the values of both operands are on stack and substitution is performed by ($Beta$).

It can be easily noticed that SECD machine transfer rules correspond one to one with the semantic equations in lifted continuation form and *Stack* corresponds to continuations in semantic equations. When *Stack* is in continuation form it can be combined with *Control* into one expression, the *Control* becoming the top of *Stack*.

SECD machine always performs ($Swap$) or ($Beta$) after rules (Var) or (Abs). In continuation form rules (Var) and (Abs) may be eliminated, because it is not necessary to transfer values from *Control* to the top of *Stack* :

Algorithm 6. Reduction with continuations (tail-recursive):

$$M\ N \qquad k \longrightarrow M\ (\lambda v.v\ N\ k) \qquad (App)$$
$$m\ (\lambda v.v\ N\ k) \longrightarrow N\ (\lambda v.m\ v\ k) \qquad (Swap)$$

m – value: variable or λ-abstraction

$$n\ (\lambda v.(\lambda x.m)\ v\ k) \longrightarrow m[x := n]\ k \qquad (Beta)$$

n – value: variable or λ-abstraction.

The reduction starts with the reducible expression M applied to the initial continuation I, which corresponds to the empty stack. The successful reduction ends with the value m applied to I :

$$M\ I \twoheadrightarrow m\ I.$$

Now we come to our main observation that reduction algorithm in continuation form incrementally transforms the reducible expression itself into continuation form. The different steps are interleaved: rules (App), $(Swap)$ correspond to continuation transformation (rules (Fl), (Fr) in *Algorithm 2* and rules (Cl), (Cr) in the definition of β-reduction) and rule $(Beta)$ corresponds to usual β-reduction step.

What happens if expression is transformed to continuation form by *Algorithm 1* before reducing? It is clear that rules (App), $(Swap)$ become unnecessary and can be eliminated. Rule $(Beta)$ must be slightly changed because the continuation of a λ-abstraction must be concatenated to the current continuation. But concatenation is also performed by β-reduction, thus rule $(Beta)$ is a double β-reduction. The rule for values $(Var - Abs)$ applies continuation to the value.

Algorithm 7. Reduction of expressions with continuations:

$$(\lambda x k'.M)\ N\ K \longrightarrow M[x := N,\quad k' := K] \qquad (Beta)$$
$$M\ (\lambda v.S) \longrightarrow S[v := M],\quad M \text{ – value} \qquad (Var - Abs)$$

Algorithm 6 and 7 produce the same result for expression M and its continuation transformation $\overline{M}$ correspondingly:

$$M\ I \underset{6}{\twoheadrightarrow} m\ I \Longleftrightarrow \overline{M}\ I \underset{7}{\twoheadrightarrow} m\ I.$$

If expression M is ***closed,*** i.e. does not contain free variables, rule $(Var - Abs)$ becomes unnecessary and $\overline{M}$ can be

reduced to its value by pure β-reduction.

Conclusions. We have investigated continuation transformation and its relation to reduction algorithms. Reduction algorithms with explicit control perform incremental continuation transformation. This observation was the basis for developing the reduction algorithm of functions in continuation form.

Continuation transformation is related to *flattening* and the reduction algorithm of transformed expressions corresponds to *resolution*, used to implement equational programs (Bellia and Levy, 1986).

Continuations are also related to *difference lists*, used in Prolog programming, because they both contain a variable in the list's tail.

REFERENCES

Bellia, M., and G.Levy (1986). The relation between logic and functional languages: a survey. *The Journal of Logic Programming*, **3**, 217–236 .

Felleisen, M., and D.P.Friedman (1986). *Control Operators, the SECD-Machine, and the λ-Calculus.* Indiana University TR, No.197.

Meyer, A.R., and M.Wand (1978). Continuation Semantics in Typed Lambda-Calculi. In *Lecture Notes in Computer Science.* Springer.

Strachey, C., and C.P.Wadsworth (1974). *Continuations: A Mathematical Semantics for Handling Full Jumps.* Oxford University PRG–11.

Wand, M., and D.P.Friedman (1978). Compiling lambda-expressions using continuations and factorizations. *Computer languages*, **3**, 241–263.

Wand, M (1980). Continuation-based program transformation strategies. *JACM*, **27**(1), 164–180 .

Received September 1990

K. Urbaitis received the Degree of Candidate of Technical Sciences from the Institute of Cybernetics of Estonian Academy of Sciences in 1988. He is a researcher at the Department of Management Systems in the Institute of Mathematics and Informatics of Lithuanian Academy of Sciences. His research interests include the foundations and integration of higher-order programming paradigms and languages.