J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press

THEORETICAL PEARLS

CPS in Little Pieces: Composing Partial Continuations

Daniel P. Friedman [†]

Amr Sabry [‡]

Computer Science Department, Indiana University
Bloomington, IN 47405, USA

Abstract

This paper presents a new two-stage CPS algorithm. The first stage plants trivial partial continuations via a recursive-descent traversal and the second stage is a rewrite system that terminates when all calls are tail calls. The algorithm combines the metaphors of the Plotkin-style CPS transformation along with reduction in the λ-calculus.

1 Introduction

The CPS transformation is usually presented as a recursive-descent algorithm that constructs continuations on the fly (Plotkin, 1976; Fischer, 1993). Instead we tease the transformation into two parts: a simple recursive-descent traversal that plants trivial partial continuations, and a rewriting system for composing and simplifying these partial continuations.

The remainder of this section illustrates these ideas with two examples written in the untyped call-by-value λ-calculus. The examples characterize the operational aspects of the two parts of the transformation, but the untyped framework fails to capture some essential invariants. Hence Sections 2 and 3 formalize our algorithms in the context of a typed CPS language with partial continuations. Section 4 is some discussion of the two algorithms and the conclusion, Section 5, puts our intuition into context.

If we work in the untyped call-by-value λ-calculus, the recursive-descent part of the transformation translates a term e to $\lambda k.ke^*$ where:

$$\begin{array}{rcl} x^* & = & x \\ (\lambda x.e)^* & = & \lambda k.\lambda x.ke^* \\ (e_1 e_2)^* & = & e_1^*\ I\ e_2^* \end{array}$$

[†] Supported by the National Science Foundation under Grant No. CCR-9987458.
[‡] Supported by the National Science Foundation under Grant No. CCR-0196063.

Every application takes I as a trivial partial continuation. Every λ takes an additional continuation argument that is immediately applied to the body. For example, the term $x(\lambda y.y)z$ translates to $\lambda k.k((x\ I_1\ (\lambda k.\lambda y.ky))\ I_2\ z)$, where we have subscripted the two occurrences of I for clarity.

The second part of the transformation processes each nontail call by composing its surrounding context with its partial continuation. For example, the call $(x\ I_1\ (\lambda k.\lambda y.ky))$ has k as a context and I_2 as a partial continuation. Merging these produces $\lambda k.(x\ I_1\ (\lambda k.\lambda y.ky))\ k\ z$.

The remaining nontail call is in an application position expecting k and z as arguments. This context is merged with the partial continuation I_1 to produce $\lambda k.x(\lambda v.vkz)(\lambda k.\lambda y.ky)$, which is the result of the traditional CPS transformation.

As another example, consider the term $x(y(zw))$. Applying the first algorithm produces $\lambda k.k(xI_1(yI_2(zI_3w)))$. One reduction sequence for the second algorithm is:

$$\begin{array}{ll} & \lambda k.k(xI_1(yI_2(zI_3w))) \\ \to & \lambda k.k(xI_1(z(\lambda v_2.yI_2v_2)w)) \\ \to & \lambda k.xk(z(\lambda v_2.yI_2v_2)w) \\ \to & \lambda k.z(\lambda v_1.xk(yI_2v_1))w \\ \to & \lambda k.z(\lambda v_1.y(\lambda v_2.xkv_2)v_1)w \end{array}$$

In the first line, the return value of z is passed to y, whose return value is then passed to x. Thus for example, it would be possible to evaluate the term in a context that binds z to a function that ignores its continuation and returns an `int`, binds y to a function taking an `int` as its second argument and returning a `bool`, and binds x to a function taking a `bool` as its second argument. After two reduction steps, in the third line, the return value of z is passed directly to x since the call to y has been absorbed as part of the continuation of z. Since z ignores its continuation and returns an `int`, the term is ill-typed. Given sensible restrictions on the free variables, the term becomes well-typed and reduction can be shown to preserve typing. To make this explicit, we move to a typed calculus.

2 Source and Target Typed Languages

The types and terms of the source calculus are:

$$\begin{array}{lll} t & ::= & b \mid t \to t \\ e & ::= & x \mid \lambda x.e \mid ee \end{array}$$

where b ranges over an unspecified collection of base types.

The presence of partial continuations in the CPS language complicates the types of the language. As Filinski argues (1999; 1996), the main challenge in typing partial continuations is in defining a rich enough answer type in which all types can be embedded and projected. Our typed framework adapts the development of Sabry (1996) for a language with control primitives that can express partial continuations, and has similarities to Filinski's presentation but is considerably simpler since we are not concerned with the operational interpretation of the embeddings and projections. The types of the CPS language include a type o of answers, as well as polymorphic types:

$$u \quad ::= \quad b \mid o \mid u \to u \mid \alpha \mid \forall\alpha.u$$

Each partial continuation will return a value of type o. In order to compose partial continuations with regular functions, two polymorphic constants are included to mediate between the type o and the types expected by the context of the partial continuation. The set of CPS terms is:

$$\begin{array}{llcl}
\textit{(Values)} & W & ::= & x \mid \lambda k.K \mid \#^u A \\
\textit{(Partial Continuations)} & K & ::= & k \mid \lambda x.A \mid WK \mid @^u \\
\textit{(Partial Answers)} & A & ::= & KW \\
\textit{(Programs)} & P & ::= & \lambda k.A
\end{array}$$

Ignoring the polymorphic constants prompt (#) and abort (@) for the moment, the CPS language is traditional (Sabry & Felleisen, 1993). Values are either variables or continuation transformers. Continuations are either variables, functions mapping values to answers, or the result of transforming another continuation. All continuations must have type $u \to o$ for some u. Answers are produced by applying continuations and must be of type o. Entire programs abstract over the single free continuation variable k and return an answer.

The constant @ has type $\forall\alpha.\alpha \to o$, and must be applied to a type u before it is used. It embeds a value into the answer type that is common to all continuations. The constant # has type $\forall\alpha.o \to \alpha$, and must also be first applied to a type u. It projects values that were embedded into the answer type back to their original type. Both constants mediate between answers and values as will be apparent in the next section when we present the modified recursive-descent algorithm. If these constants are never used, then all calls are of the form WKW, *i.e.*, all calls are tail calls.

As usual, the semantics of the CPS language is given by β and η reductions. In addition, the prompt and abort at identical types are stipulated to be inverses:

$$\begin{array}{rcll}
\#^u(@^u W) & \to & W & \\
@^u(\#^u A) & \to & A & \\
(\lambda k.K_1)K_2 & \to & K_1[k := K_2] & \\
(\lambda x.A)W & \to & A[x := W] & \\
\lambda k.Wk & \to & W & \text{where } k \notin FV(W) \\
\lambda x.Kx & \to & K & \text{where } x \notin FV(K)
\end{array}$$

All but the β and η reductions on source terms are considered to be administrative. As expected the rules are type-preserving.

3 Composing Partial Continuations

In this section, we modify the recursive-descent algorithm to use the type judgments of the source language. By introducing prompts and aborts at the appropriate types in the output of the translation, it becomes evident that both phases of the new CPS algorithm are type-preserving.

As usual (Meyer & Wand, 1985), the CPS translation on types is the following:

$$\begin{array}{rcl} b^* & = & b \\ (t_1 \rightarrow t_2)^* & = & (t_2^* \rightarrow o) \rightarrow t_1^* \rightarrow o \end{array}$$

The revised recursive-descent algorithm treats variables and procedures as in the untyped case. For applications, the trivial partial continuation I is replaced with an @ that embeds the return type of the application into the answer type for continuations. The entire application is surrounded with a # that extracts the value from the answer type:

$$\begin{array}{rcl} x^* & = & x \\ (\lambda x.e)^* & = & \lambda k.\lambda x.ke^* \\ (e_1^{t_2 \rightarrow t} e_2)^* & = & \#^{t^*}(e_1^* \ @^{t^*} \ e_2^*) \end{array}$$

The type t_2 mentioned in the left-hand side of the last equation does not appear in the right-hand side because we are only interested in the return type.

It is interesting that even the trivial planting of partial continuations produces terms that appear to have been transformed to CPS (as far as the types are concerned).

Proposition 3.1
If $\Gamma \vdash e : t$ then $\Gamma^* \vdash e^* : t^*$.

Then the second pass can be described as "Do any of the administrative reductions on CPS terms and any of the following two reductions for composing partial continuations."

$$\begin{array}{rcll} K(\#^u(WK_1W_1)) & \rightarrow & W(\lambda v.K(\#^u(K_1v)))W_1 & \textit{(lift arg)} \\ (\#^u(WK_1W_1))K_2W_2 & \rightarrow & W(\lambda v.(\#^u(K_1v))K_2W_2)W_1 & \textit{(lift fun)} \\ & & \text{where } u = (u_1 \rightarrow o) \rightarrow u_2 \rightarrow o & \end{array}$$

The first rule *(lift arg)* has $\#^u(WK_1W_1)$ as the *arg*ument to K and the second rule *(lift fun)* has $\#^u(WK_1W_1)$ as the *fun*ction applied to K_2 and W_2. In the first rule, the invocation of W is a nontail call, but in the right-hand side the invocation of W is a tail call. Thus, we reduce the left-hand expression by one nontail call. Similarly, for the second rule, since the invocation of W is a nontail call, which becomes a tail call.

Proposition 3.2
For each rule if the left-hand side has type t, then so does the right-hand side.

4 Discussion

Our goal is to make all W-calls be tail calls. In other words, we want all calls to be of the form (WKW_1). We can see that the rules *(lift arg)* and *(lift fun)* do just that. The only difference in these two rules in the right-hand side is what the newly created K looks like. In both cases, however, the new K is formed by composing the context of (WK_1W_1) from the left-hand side and K_1. Clearly each lift rule removes one nontail call and the CPS target language guarantees that no additional nontail calls are introduced. Therefore, our goal is met.

When we first formalized the algorithm, it contained three rules, but by re-arranging the argument order, so that the continuation was the first instead of the second argument, one of the three rules became a special case of one of the two remaining rules. In addition, we got the *let*-optimization as a byproduct. (By *let*-optimization, we mean an additional rule for translating source judgments that special-cases the application rule when e_1 is a λ-expression. The inclusion of the administrative reductions obviates this special-casing.)

Regrettably, first class continuations do not fall out of this approach. If we wish to include a control operator such as $\mathcal{C}$ (Felleisen *et al.*, 1987) in our language, then we must process its argument and when there are no more tail calls, introduce another pass to finally start removing all occurrences of $\mathcal{C}$ just as we would for all existing CPS algorithms.

5 Conclusion

From an intuitive perspective, we can see that the abort partial continuation absorbs the surrounding context up to the prompt, a bit at a time until the entire surrounding context has been consumed. Unlike conventional (Plotkin-style) CPS algorithms, which do not introduce partial continuations, ours has this bit of overhead. But these partial continuations disappear from the target-language output if they are not present in the source language.

Because each transformation not only preserves types but also preserves correctness, we know that once the first algorithm completes, we can apply the CPS semantics to reduce to the same final value after each reduction step of the second algorithm. As a result, we know that once no more lift rules apply, not only are we free of nontail calls, but we know that we have correctly preserved the semantics of the original program.

References

Felleisen, Matthias, Friedman, D. P., Kohlbecker, E., & Duba, B. (1987). A syntactic theory of sequential control. *Theoretical computer science*, **52**(3), 205–237. Preliminary version: Reasoning with Continuations, in Proceedings of the 1st IEEE Symposium on Logic in Computer Science, 1986.

Filinski, Andrzej. 1996 (May). *Controlling effects.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Filinski, Andrzej. 1999 (Jan.). Representing layered monads. *Pages 175–188 of: Conference record of POPL 99: The 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages.* ACM, New York, NY.

Fischer, Michael J. (1993). Lambda-calculus schemata. *Lisp and symbolic computation*, **6**(3/4), 259–288.

Meyer, A. R., & Wand, M. (1985). Continuation semantics in typed lambda-calculi. *Pages 219–224 of: Proceedings workshop logics of programs.* Lecture Notes in Computer Science, 193.

Plotkin, G. (1976). Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, **1**(1), 125–159.

Sabry, Amr. 1996 (March). *Note on axiomatizing the semantics of control operators*. Tech. rept. CIS-TR-96-03. Department of Computer and Information Science, University of Oregon.

Sabry, Amr, & Felleisen, Matthias. (1993). Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, **6**(3–4), 289–360.