Lecture Notes on Continuation-Passing Style

15-417/817: HOT Compilation
Frank Pfenning

Lecture 20
April 8, 2025

1 Introduction

Continuations are a generally useful technique in functional programming. For example, during search we might pass a success continuation that encapsulates what needs to be done after the current subgoal succeeds, or a failure continuation that encapsulates what need to be tried in case the current subgoal fails. The type of a search function often looks like

$$\mathit{search}\ (p : \mathsf{problem})\ (\mathit{sc} : \mathsf{data} \to \alpha)\ (\mathit{fc} : \alpha) : \alpha$$

where data is the information about the solution to the subproblem passed on the to success continuation sc.

There is also the notion of *first-class continuation* which provides a programmatic way to capture the remainder of the computation. If we can pass this around we can jump back to an earlier state of the computation to give us a flexible way, for example, to do backtracking. On the other hand, first-class continuations are a *control effect*, which means many techniques for reasoning about the behavior of functional programs are no longer sound. Functions may not only return a value or diverge, they may also invoke the continuation and jump to an entirely different place in the computation.

We can use (second-class, that is, just functions) continations as functions in defining an interpreter for a programming language [Reynolds, 1972]. For example, if $E : \mathit{exp}$ is an expression of a source language with functions, we might define

$$\mathit{eval}\ (E : \mathit{exp})\ (K : \mathit{val} \to \mathit{val}) : \mathit{val}$$

with

$$\begin{array}{l}\mathit{eval}\ E\ K = \mathbf{match}\ E\ \mathbf{with} \\ \quad \mid\ (\mathit{app}\ E_1\ E_2) \Rightarrow \mathit{eval}\ E_1\ (\lambda V_1.\ \mathit{eval}\ E_2\ (\lambda V_2.\ \mathit{apply}\ V_1\ V_2\ K)) \\ \quad \ldots\end{array}$$

where apply evaluates the result of substituting V_2 for the variable abstracted over in V_1 with continuation K.

We have seen this technique in Lecture 16 when defining a translation from ND to Sax that respects bidirectional typing. We can also adapt this to the situation here: translating from a functional source language to an (also functional) target language in which certain aspects of the

language (such call-by-value vs. call-by-name, or left-to-right vs right-to-left evaluation order) are made explicit. This is the idea behind the CPS transformation.

It turns out that the transformation into continuation-passing style is very closely related to our translation into destination-passing style (from ND, our functional source language, to Sax, our imperative target language). In the case of destination-passing style, every function is given an additional argument which is the destination for its result. In continuation-passing style, every function is given an additional argument which is the function to call on the result. I suspect there is a not-too-difficult theorem that formally relates these two translations; here we just rely on intuition.

2 Functions

In much of programming language theory, functions (or more generally expressions of negative type) are seen as the "good and elegant case" and data such as pairs or sums are the "bad and ugly case" because necessity of introducing matching constructs. This judgment is bolstered by the elegance of the λ-calculus (whether typed or untyped). For example, the addition of positive types makes characterizing equality on expressions much more difficult. Type systems based on the sequent calculus (including its semi-axiomatic form) are more equitable and in a sense closer to models of computations such as Turing machines. Perhaps the dichotomy between functional (Church's λ-calculus) and imperative (Turing machines) notions of computation can be recast as the tension between negative (Church) and positive (Turing) types.

In any case, in much of the work on continuation-passing transformation there is a strong focus on functions. This is also the case in the technical report Danvy and Pfenning [1995] upon which this lecture is based. We define a transformation

$$\|e\|\ \underline{K} = e'$$

where e is a source expression, $\underline{K}$ is a metalavel continuation, and e' is an expression in continuation-passing style.

There are different translations of this kind, depending on whether we want to model call-by-value or call-by-name, left-to-right or right-to-left, and whether continuation are passed to functions first or last. We describe the call-by-value, left-to-right, continuations-last approach.

We start with functions. They take an extra argument, which is they continuation to call with their value instead of returning it. Also, a λ-expression *is* a value, so we pass the translated form to the continuation.

$$\|\lambda x.\, e\|\ \underline{K} \quad = \quad \underline{K}\,(\lambda x.\, \lambda k.\ \|e\|\ (\underline{\lambda} t.\, k\, t))$$

Let's examine the parallels with our translation from ND to Sax:

$$[\![\lambda x.\, e]\!]\, d \quad = \quad \mathbf{write}\ d\ ((x, y) \Rightarrow [\![e]\!] y)$$

- Applying $\underline{K}$ corresponds to writing to d.
- The extra variable y (which is destination for the function) corresponds to the continuation variable k.
- Translating the body of the function with continuation k corresponds to translation the body of the function with destination y.

Now we look at application. Recall:

$$\begin{array}{rcll} [\![e_1\, e_2]\!]\, d & = & \mathbf{cut}\ t_1 & \%\ \textit{create destination for function} \\ & & \quad [\![e_1]\!]\, t_1 & \%\ \textit{write function to } t_1 \\ & & \mathbf{cut}\ t_2 & \%\ \textit{create destination for argument} \\ & & \quad [\![e_2]\!]\, t_2 & \%\ \textit{write argument value to } t_2 \\ & & \mathbf{read}\ t_1\ (t_2, d) & \%\ \textit{apply function to value, writing result to } d \end{array}$$

Translating this to continuation-passing:

- Instead of creating fresh destinations t_1 and t_2 we create fresh continuations, taking t_1 and t_2 as arguments, respectively.
- Instead of passing the pair (t_1, d), we use the curried form, passing first t_2 and then the original continuation $\underline{K}$ (in η-expanded form in order to allow the object-level abstraction λv).

$$\|e_1\, e_2\|\ \underline{K} \quad = \quad \|e_1\|\ (\underline{\lambda} t_1.\ \|e_2\|\ (\underline{\lambda} t_2.\, t_1\, t_2\, (\lambda v.\, \underline{K}(v))))$$

One important observation is that the Sax code is explicitly parallel (due to the nature of the cut), while the CPS translation is explicitly sequential. That's because the purposes are different: CPS is supposed to fix evaluation order, but translation to Sax actually uncovers latent parallelism.

To complete the translation, variables stand for values so they are just passed to the continuation.

$$\|x\|\ \underline{K} \quad = \quad \underline{K}(x)$$

This corresponds to

$$[\![x]\!]\, d \quad = \quad \mathbf{id}\ d\ x$$

Here is a summary of the translation:

$$\begin{array}{rcl} \|x\|\ \underline{K} & = & \underline{K}\,(x) \\ \|\lambda x.\, e\|\ \underline{K} & = & \underline{K}\,(\lambda x.\, \lambda k.\ \|e\|\ (\underline{\lambda} t.\, k\, t)) \\ \|e_1\, e_2\|\ \underline{K} & = & \|e_1\|\ (\underline{\lambda} t_1.\ \|e_2\|\ (\underline{\lambda} t_2.\, t_1\, t_2\, (\lambda v.\, \underline{K}(v)))) \end{array}$$

3 A Small Example

Let's translate

$$\lambda x.\, f\, x\, (g\, x)$$

for some variables f and g. We want the translation to return a function in continuation-passing style, so at the top level we call it with the continuation $\underline{\lambda} t.\, t$. When the focus of the step is embed-

ded, we highlight it in blue.

$$
\begin{aligned}
& \|\lambda x.\, f\, x\, (g\, x)\|\ (\underline{\lambda}t.\, t) \\
= \ & (\underline{\lambda}t.\, t)\ (\lambda x.\, \lambda k.\ \|f\, x\, (g\, x)\|\ \underline{\lambda}t.\, k\, t) \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\, (g\, x)\|\ \underline{\lambda}t.\, k\, t \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\|\ (\underline{\lambda}t_1.\ \|g\, x\|\ (\underline{\lambda}t_2.\, t_1\, t_2\, (\lambda v.\, (\underline{\lambda}t.\, k\, t)\, v))) \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\|\ (\underline{\lambda}t_1.\ \|g\, x\|\ (\underline{\lambda}t_2.\, t_1\, t_2\, (\lambda v.\, k\, v))) \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\|\ (\underline{\lambda}t_1.\ \|g\|\ (\underline{\lambda}t_3.\ \|x\|\ (\underline{\lambda}t_4.\, t_3\, t_4\, (\lambda w.\, (\underline{\lambda}t_2.\, t_1\, t_2\, (\lambda v.\, k\, v))\, (w))))) \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\|\ (\underline{\lambda}t_1.\ \|g\|\ (\underline{\lambda}t_3.\ \|x\|\ (\underline{\lambda}t_4.\, t_3\, t_4\, (\lambda w.\, t_1\, w\, (\lambda v.\, k\, v))))) \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\|\ (\underline{\lambda}t_1.\ \|g\|\ (\underline{\lambda}t_3.\, t_3\, x\, (\lambda w.\, t_1\, w\, (\lambda v.\, k\, v)))) \\
= \ & \lambda x.\, \lambda k.\ \|f\, x\|\ (\underline{\lambda}t_1.\, g\, x\, (\lambda w.\, t_1\, w\, (\lambda v.\, k\, v))) \\
= \ & \lambda x.\, \lambda k.\ \|f\|\ (\underline{\lambda}t_5.\ \|x\|\ (\underline{\lambda}t_6.\, t_5\, t_6\, (\lambda u.\, (\underline{\lambda}t_1.\, g\, x\, (\lambda w.\, t_1\, w\, (\lambda v.\, k\, v)))\, u))) \\
= \ & \lambda x.\, \lambda k.\ \|f\|\ (\underline{\lambda}t_5.\ \|x\|\ (\underline{\lambda}t_6.\, t_5\, t_6\, (\lambda u.\, g\, x\, (\lambda w.\, u\, w\, (\lambda v.\, k\, v))))) \\
= \ & \lambda x.\, \lambda k.\, f\, x\, (\lambda u.\, g\, x\, (\lambda w.\, u\, w\, (\lambda v.\, k\, v)))
\end{aligned}
$$

We can understant the final line as saying:

1. The translated function takes x and a continuation k as argument.
2. Evaluate $f\, x$, call the result u.
3. Evaluate $g\, x$, call the result w.
4. Apply u to w, call the result v.
5. Pass v to k.

This is a perfectly sensible interpretation what the initial function should do in continuation-passing style.

4 Typing

Without giving it much thought, we would give the translation the meta-level type

$$\|-\| : \mathsf{exp} \to (\mathsf{val} \to \mathsf{val}) \to \mathsf{val}$$

As often for continuation-passing style, the result type val is fixed only because of the chosen initial continuation. More generally, the translation proper would have type

$$\|-\| : \mathsf{exp} \to (\mathsf{val} \to \tau) \to \tau$$

for the so-called final answer type τ. This misses the fact that the continuation $(\mathsf{val} \to \tau)$ is used linearly, so with more precision we could write

$$\|-\| : \mathsf{exp} \to (\mathsf{val} \to \tau) \multimap \tau$$

Actually, come to think of it, the input expression is analyzed linearly and so are the intermediate values, so we might have

$$\|-\| : \mathsf{exp} \multimap (\mathsf{val} \multimap \tau) \multimap \tau$$

This strong typing doesn't translate fully to typing of the object language. If $\Gamma \vdash e : A$ then $\Gamma^*, k : A^* \to \tau \vdash \|e\|\ (\underline{\lambda} t.\, k\, t) : \tau$ where we define

$$(A \to B)^* = A^* \to (B^* \to \tau) \to \tau$$

and base types remain unchanged.

In this translation, the continuation variable k is used *linearly*. In fact, we have a stronger property namely that at any point in the translation there is exactly one continuation variable k in scope, which will be used exactly once. This property is taken advantage of in the Standard ML of New Jersey compiler which does translate the source to an intermediate form in continuation-passing style. It is confirmed by the relationship to destination passing style: the destination is in the succedent of the typing judgment, which is always a singleton. For the continuation, this is not enforced at the level of judgments in a similar way.

We can capture the linearity part of this observation in the type translation:

$$(A \to B)^* = A^* \to (B^* \to \tau) \multimap \tau$$

On the other hand, the functions from A^* and B^* are not necessarily linear because they represent functions at the object level. Now if those were typed, say, linearly, then their translations would also be linear (and the same for other substructural properties).

This linearity property (and some variations thereof) where noted by Berdine et al. [2002]. Later is was noted by Danvy [1994] that the temporary variables introduced by the CPS translation satisfy an ordering property. Terms satisfying this property can be translated back to direct style. At this time we have not investigated if this ordering among the temporary variables can be captured using (possible adjoint) ordered types.

5 Other Constructs

Knowing the compilation from ND to Sax, the translation of other constructs is not difficult. For negative types, we have to introduce a new continuation for the embedded expressions while positive types are simpler. Starting with lazy records.

$$\begin{aligned} \|\{\ell \Rightarrow e_\ell\}_{\ell \in L}\|\ \underline{K} &= \underline{K}\,(\{\ell \Rightarrow \lambda k.\ \|e_\ell\|\ (\underline{\lambda} t.\, k\, t)\}_{\ell \in L}) \\ \|e.\ell\|\ \underline{K} &= \|e\|\ (\underline{\lambda} t.\, t.\ell\,(\lambda v.\, \underline{K}(v))) \end{aligned}$$

And eager pairs.

$$\begin{aligned} \|(e_1, e_2)\|\ \underline{K} &= \|e_1\|\ (\underline{\lambda} t_1.\ \|e_2\|\ (\underline{\lambda} t_2.\, \underline{K}\,(t_1, t_2))) \\ \|\mathbf{match}\ e\ ((x, y) \Rightarrow e')\|\ \underline{K} &= \|e\|\ (\underline{\lambda} t.\, \mathbf{match}\ t\ ((x, y) \Rightarrow \|e'\|\ \underline{K})) \end{aligned}$$

We'll leave it at that, since there is not particular difficulty. It is a useful exercise to remind yourself of our compilation and recognize the parallels in the patterns.

One thing that should be noted is that while there is a strong analogy between the continuation-passing translation and translation to destination-passing style, in some ways continuations are a lot more powerful because of what they allow us to do with a value returned. For example, we can build in certain classes of exceptions by using a (linear, lazy) pair of continuations, one for success and one for the exception [Berdine et al., 2002]. Or we can go so far as **callcc** as mentioned in the introduction. The latter will not use its continuations linearly, and as such adds essential expressive power, but at the expense of writing code in an effectful language that is much more difficult to reason about that a pure functional language.

References

Josh Berdine, Peter O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15:181–208, September 2002.

Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical Report CMU-CS-95-121, Department of Computer Science, Carnegie Mellon University, February 1995.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.