Delimited continuations

\[ \lambda x.\langle \ldots \langle \ldots Sk.M \ldots \rangle \ldots \rangle \]

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
Delimited continuations *by example*
### Values

$V ::= x \quad \text{variable} \\
| \lambda x. M \quad \text{abstraction}$

### Terms

$L, M ::= V \quad \text{value} \\
| L \ M \quad \text{application} \\
| \langle M \rangle \quad \text{reset} \\
| S \ k.M \quad \text{shift}$

### Reductions

\[
\begin{align*}
E(\lambda x. M) \ V & \rightsquigarrow E[M\{V/x\}] \\
E(\langle V \rangle) & \rightsquigarrow E[V] \\
E[E_2[S \ k.M]] & \rightsquigarrow E[M\{(\lambda y. (E_2[y]))/k\}] \\
\end{align*}
\]

### Alternative operators

\[
\begin{align*}
E(\langle E_2[S_0 \ k.M] \rangle) & \rightsquigarrow E[M\{(\lambda y. (E_2[y]))/k\}] \\
E(\langle E_2[F \ k.M] \rangle) & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}] \\
E(\langle E_2[F_0 \ k.M] \rangle) & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}] \\
\end{align*}
\]
Basics

Example

Reading

Values

\[ V ::= x \text{ variable} \mid \lambda x. M \text{ abstraction} \]

Terms

\[ L, M ::= V \text{ value} \mid L M \text{ application} \mid \langle M \rangle \text{ reset} \mid S k.M \text{ shift} \]

Contexts

\[ E[\cdot] ::= [\cdot] \mid E[\cdot] M \mid E[V[\cdot]] \mid E[\langle \cdot \rangle] \]

Reductions

\[
\begin{align*}
E(\lambda x. M) V & \rightsquigarrow E[M[V/x]] \\
E[\langle V \rangle] & \rightsquigarrow E[V] \\
E[\langle E_2[S \ k. M] \rangle] & \rightsquigarrow E[\langle M\{(\lambda y. E_2[y])\}/k\rangle]\]
\end{align*}
\]

Alternative operators

\[
\begin{align*}
E[\langle E_2[S_0 \ k. M] \rangle] & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}] \\
E[\langle E_2[F \ k. M] \rangle] & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}] \\
E[\langle E_2[F_0 \ k. M] \rangle] & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}] \\
\end{align*}
\]
### Basics

#### Values

| \( V \) | \( x \) | variable | \( \lambda x. M \) | abstraction |

#### Terms

| \( L, M \) | \( V \) | value | \( L M \) | application | \( \langle M \rangle \) | reset | \( S k.M \) | shift |

#### Contexts

\[ E[\cdot] \ ::= \ [\cdot] \quad E[\cdot\ M] \quad E[V[\cdot\]] \quad E[\langle\cdot\rangle] \]

#### Reductions

\[
\begin{align*}
E(\lambda x. M) V & \rightsquigarrow E[M\{V/x\}] \\
E[\langle V \rangle] & \rightsquigarrow E[V] \\
E[\langle E_2[S k.M] \rangle] & \rightsquigarrow E[\langle M\{(\lambda y. E_2[y])/k\}\rangle]
\end{align*}
\]

#### Alternative operators

\[
\begin{align*}
E[\langle E_2[S_0 k.M] \rangle] & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}] \\
E[\langle E_2[F k.M] \rangle] & \rightsquigarrow E[\langle M\{(\lambda y. E_2[y])/k\}\rangle] \\
E[\langle E_2[F_0 k.M] \rangle] & \rightsquigarrow E[M\{(\lambda y. E_2[y])/k\}]
\end{align*}
\]
\[ E[\langle E_2[S \cdot k \cdot M] \rangle] \quad \leadsto \quad E[\langle M\{\lambda y.\langle E_2[y] \rangle\}/k \rangle] \]
Rule: $E[\langle E_2[S \ k.\ M]\rangle] \rightsquigarrow E[\langle M\{\langle \lambda y.\langle E_2[y]\rangle\}/k\}\rangle]$ 

Program

$\langle 1 + \langle (S \ k_1.\ k_1 \ 100 + k_1 \ 10) + S k_2.\ S k_3.\ 1\rangle \rangle$ 

Decompose

$E = \langle 1 + \ldots \rangle$ 
$E_2 = \ldots + S k_2.\ S k_3.\ 1$ 
$M = k_1 \ 100 + k_1 \ 10$ 

Substitute

$M\{\langle \lambda y.\langle E_2[y]\rangle\}\} = (k_1 \ 100 + k_1 \ 10)\{\langle \lambda y.\langle y + S k_2.\ S k_3.\ 1\rangle\}/k_1\}$ 
$= \ldots$ 
$= \langle 100 + S k_2.\ S k_3.\ 1\rangle + \langle 10 + S k_2.\ S k_3.\ 1\rangle$ 

Reconstruct

$E[\langle M\{\langle \lambda y.\langle E_2[y]\rangle\}/k\}\rangle] = \langle 1 + \langle 100 + S k_2.\ S k_3.\ 1\rangle + \langle 10 + S k_2.\ S k_3.\ 1\rangle\rangle$
Example

Rule: \( E[\langle E_2[S\,k.M]\rangle] \leadsto E[\langle M\{\lambda y.\langle E_2[y]\rangle\}/k\rangle] \)

Program: \( \langle 1 + \langle 100 + S\,k_2.S\,k_3.1 \rangle + \langle 10 + S\,k_2.S\,k_3.1 \rangle \rangle \)

Decompose:
\[
\begin{align*}
E &= \langle 1 + \langle \cdot + \langle 10 + S\,k_2.S\,k_3.1 \rangle \rangle \rangle \\
E_2 &= 100 + \langle \cdot \\
M &= S\,k_3.1
\end{align*}
\]

Substitute:
\[
M\{\lambda y.\langle E_2[y]\rangle\} = S\,k_3.1
\]

Reconstruct:
\[
E[\langle M\{\lambda y.\langle E_2[y]\rangle\}/k\rangle] = \langle 1 + \langle S\,k_3.1 \rangle + \langle 10 + S\,k_2.S\,k_3.1 \rangle \rangle
\]
Rule: \[ E[\langle E_2[S\,k\,M]\rangle] \rightsquigarrow E[\langle M\{ (\lambda y. \langle E_2[y]\rangle)/k \} \rangle] \]

Program \[ \langle 1 + \langle \langle S \, k_3.1 \rangle + \langle 10 + S \, k_2.\, S \, k_3.1 \rangle \rangle \rangle \]

Decompose
\[
\begin{align*}
E &= \langle 1 + \langle \, + \langle 10 + S \, k_2.\, S \, k_3.1 \rangle \rangle \rangle \\
E_2 &= [ \, ] \\
M &= 1
\end{align*}
\]

Substitute
\[ M\{ (\lambda y. \langle E_2[y]\rangle) \} = 1 \]

Reconstruct
\[ E[\langle M\{ (\lambda y. \langle E_2[y]\rangle)/k \} \rangle] = \langle 1 + \langle 1 \rangle + \langle 10 + S \, k_2.\, S \, k_3.1 \rangle \rangle \]
Rule:  \[ E[\langle E_2[S \cdot M]\rangle] \rightsquigarrow E[\langle M\{\lambda y.\langle E_2[y]\rangle\}/k\rangle]\]

Program  \[ \langle 1 + \langle 1 \rangle + \langle 10 + S \cdot S k_3.1 \rangle \rangle \]

Decompose
\[
\begin{align*}
E & = \langle 1 + \langle 1 \rangle + \cdot \rangle \\
E_2 & = \langle 10 + \cdot \rangle \\
M & = S k_3.1
\end{align*}
\]

Substitute
\[ M\{\lambda y.\langle E_2[y]\rangle\} = S k_3.1 \]

Reconstruct
\[ E[\langle M\{\lambda y.\langle E_2[y]\rangle\}/k\rangle] = \langle 1 + \langle 1 \rangle + \langle S k_3.1 \rangle \rangle \]
Rule: \[ E[\langle E_2[S \, k. \, M] \rangle] \leadsto E[\langle M\{\lambda y.\langle E_2[y]\rangle\}/k\rangle] \]

Program: \[ \langle 1 + \langle 1 \rangle + \langle S \, k_3.1 \rangle \rangle \]

Decompose:
\[
\begin{align*}
E &= \langle 1 + \langle 1 \rangle + \cdots \rangle \\
E_2 &= \langle 1 \rangle \\
M &= 1
\end{align*}
\]

Substitute:
\[ M\{\lambda y.\langle E_2[y]\rangle\} = 1 \]

Reconstruct:
\[ E[\langle M\{\lambda y.\langle E_2[y]\rangle\}/k\rangle] = \langle 1 + \langle 1 \rangle + \langle 1 \rangle \rangle \]
Reading
Abstract
We describe the first implementation of multi-prompt delimited control operators in OCaml that is direct in that it captures only the needed part of the control stack. The implementation is a library that requires no changes to the OCaml compiler or runtime, so it is perfectly compatible with existing OCaml source and binary code. The library has been in fruitful practical use since 2006.

We present the library as an implementation of an abstract machine derived by elaborating the definitional machine. The abstract view lets us define a minimalistic API, scAPI, sufficient for implementing multi-prompt delimited control. We argue that a language system that supports exception and stack-overflow handling supports scAPI. With byte- and native-code OCaml systems as two examples, our library illustrates how to use scAPI to implement multi-prompt delimited control in a typed language. The approach is general and has been used to add multi-prompt delimited control to other existing language systems.

Keywords: delimited continuation, exception, semantics, implementation, abstract machine

1. Introduction
The library delimcc for delimited control for OCaml was first released at the beginning of 2006 [1] and has been used for implementing (delimited) exception handling [2].

“The delimcc library was the first direct implementation of delimited control in a typed, mainstream, mature language — it captures only the needed prefix of the current continuation, requires no code transformations, and integrates with native-language exceptions.

“The delimcc library does not modify the OCaml compiler or run-time in any way, so it ensures perfect binary compatibility with existing OCaml code and other libraries.

“Captured delimited continuations may be reinstated arbitrarily many times in different dynamic contexts.”
Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform

Tiark Rompf, Ingo Maier, Martin Odersky

Abstract
We describe the implementation of first-class and polymorphic delimited continuations in the programming language Scala. We use Scala’s pluggable typing architecture to implement a separate CPS transform. The first-class delimited continuations are implemented directly within the virtual machine. The transformed program is compiled to native code. To implement delimited continuations, we introduce a new type system, which is based on dynamic type representation. The type system allows for first-class delimited continuations, enabling the programmer to refer in types and methods to call sites. This ability makes delimited continuations strictly more powerful than their undelimited counterparts.

Keywords
Delimited continuations, selective CPS transform, pure code in direct style.

1. Introduction
Continuations, and in particular delimited continuations, are a versatile programming tool. While initially, we are interested in their ability to support ad-hoc expression codes by means of a controlled way of removing, restoring and referring to VM frames, we later turn our attention to the semantic aspects. The latter are a topic of this paper.

Classical (or full) continuations cannot be treated as a first-class construct in most modern programming languages. However, some recent systems have supported delimiters and dynamic control flow. The implementation of delimited continuations requires special provisions from the runtime system (Clinger et al. 1999), like the ability to capture and restore the run-time stack, which are not available in all environments. In particular, popular VM’s such as the JVM or the .NET CLR do not provide this low-level access to the run-time stack. One way to overcome these limitations is to support custom-defined control flow. Syntactic restrictions imposed by mandatory style can be overcome by introducing language constructs in the language level, as is done in ML, which are not supported natively. This is partly because explicit support for continuations is assumed to be part of the programming language or its type system (Biernacki et al. 2006; Shan 2007). However, in most other languages, the support for continuations is generally provided by libraries, which are not available in all environments. In particular, polymorphic VO’s such as the JVM do not provide the low-level access to the run-time stack. The low-level approach is intended to provide a way to implement delimited continuations in an environment that does not support them natively. This is achieved by a dynamic selective CPS transform, which is driven entirely by effect-annotated types and leaves pure code in direct style.

“Benchmarks indicate that this high-level approach performs competitively.

“...tackle the problem of implementing first-class continuations under the adverse conditions brought upon by the Java VM, we employ a selective CPS transform, which is driven entirely by effect-annotated types and leaves pure code in direct style.

...
“Wasm provides no direct support for non-local control flow features such as async/await, generators/iterators, lightweight threads, first-class continuations, etc. [...] compilers for source languages with such features must ceremoniously transform whole source programs in order to target Wasm [...]”

“WasmFX mechanism is based on delimited continuations extended with multiple named control tags inspired by Plotkin and Pretnar’s effect handlers [...]”

“The resume instruction consumes its continuation operand, meaning a continuation may be resumed only once — i.e., we only support single-shot continuations.”
Writing suggestions

Expressiveness
Do these implementations support multi-shot continuations?
Do these implementations support multiple prompts?
(Does either of these questions matter in practice?)

Efficiency
Under which circumstances (if any) is the performance acceptable?

Types
How are continuations typed?
Are types used in the implementations?

Usability
How usable is each approach in practice?
Schedule change next week

Tuesday 24th                LT2
3pm Monday 23rd October, FW26