Delimited continuations

$$\lambda x.\langle \ldots \langle \ldots \mathcal{S} k.M \ldots \rangle \ldots \rangle$$

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

Evaluation & the stack

Values

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

Terms

$$L, M ::= V \quad \textit{value} \quad | \quad L\ M \quad \textit{application}$$

Computation rules

$$\frac{}{(\lambda x.M)\ V \rightsquigarrow M\{V/x\}}$$

Congruence rules

$$\frac{L \rightsquigarrow L'}{L\ M \rightsquigarrow L'\ M}$$

$$\frac{M \rightsquigarrow M'}{V\ M \rightsquigarrow V\ M'}$$

Continuation-based expression evaluation

Values

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

Terms

$$L, M ::= V \quad \textit{value} \quad | \quad L\ M \quad \textit{application}$$

Continuations

$$E[\,\cdot\,] ::= [\,\cdot\,] \quad | \quad E[[\,\cdot\,]\ M] \quad | \quad E[V\,[\,\cdot\,]]$$

Computation rules

$$E[(\lambda x.M)\ V] \rightsquigarrow E[M\{V/x\}]$$

Stack-based expression evaluation

$100 + 10$

$(\lambda x.x + 2)[-]$

$1 + [-]$

$$1 + ((\lambda x.x + 2)[100 + 10])$$

110
$(\lambda x.x + 2)[-]$
$1 + [-]$

$$1 + ((\lambda x.x + 2)[110])$$

Stack-based expression evaluation

$$(\lambda x.x + 2)110$$

$$1 + [-]$$

$$1 + ([(\lambda x.x + 2)110])$$

Stack-based expression evaluation

$$110 + 2$$

$$1 + [-]$$

$$1 + [(110 + 2)])$$

Stack-based expression evaluation

112

$1 + [-]$

$$1 + [112]$$

Stack-based expression evaluation

$$1 + 112$$

$$[1 + 112]$$

113

$$113$$

Delimited continuations

The stack

Continuations

Variations & applications

Reading

Values

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

Terms

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

Continuations

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

Computation rules

$$E[(\lambda x.M)\ V] \rightsquigarrow E[M\{V/x\}]$$

$$E[\langle V \rangle] \rightsquigarrow E[V]$$

$$E[\langle E_2[\mathcal{S}\, k.M] \rangle] \rightsquigarrow E[\langle M\{(\lambda y.\langle E_2[y] \rangle)/k\} \rangle]$$

Values

V	$::=$	x	*variable*
	$\mid$	$\lambda x.M$	*abstraction*

Terms

L, M	$::=$	V	*value*
	$\mid$	$L\ M$	*application*
	$\mid$	$\langle M \rangle$	*reset*
	$\mid$	$\mathcal{S}\,k.M$	*shift*

Continuations

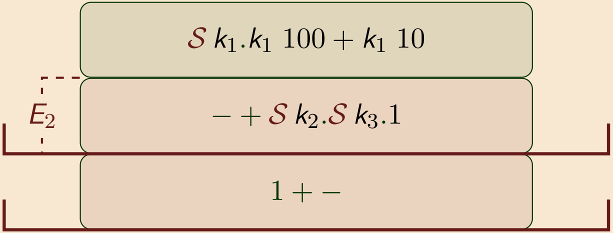
$E[\,\cdot\,]$	$::=$	$[\,\cdot\,]$
	$\mid$	$E[[\,\cdot\,]\ M]$
	$\mid$	$E[V\,[\,\cdot\,]]$
	$\mid$	$E[\langle [\,\cdot\,] \rangle]$

Computation rules

$$E[(\lambda x.M)\ V] \rightsquigarrow E[M\{V/x\}]$$
$$E[\langle V \rangle] \rightsquigarrow E[V]$$
$$E[\langle E_2[\mathcal{S}\,k.M] \rangle] \rightsquigarrow E[\langle M\{(\lambda y.\langle E_2[y] \rangle)/k\} \rangle]$$

Example

$$E[\langle E_2[\mathcal{S}\,k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$



$$\langle 1 + \langle [\mathcal{S}\ k_1.k_1\ 100 + k_1\ 10] + \mathcal{S}\ k_2.\mathcal{S}\ k_3.1\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$$(\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 100$$

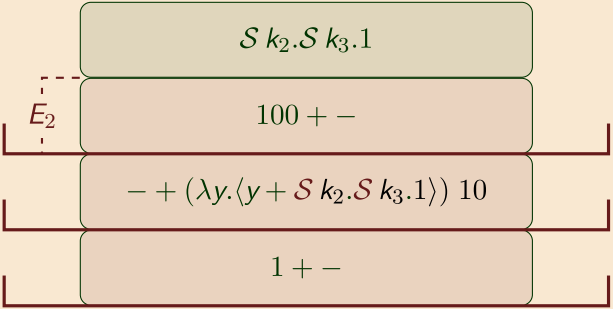
$$- + (\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 10$$

$$1 + -$$

$$\langle 1 + \langle [(\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 100] + (\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 10\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$



$$\langle 1 + \langle\langle 100 + [\mathcal{S}\, k_2.\mathcal{S}\, k_3.1]\rangle + (\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 10\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\,k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$\mathcal{S}\,k_3.1$

$- + (\lambda y.\langle y + \mathcal{S}\,k_2.\mathcal{S}\,k_3.1\rangle)\ 10$

$1 + -$

$$\langle 1 + \langle\langle[\mathcal{S}\,k_3.1]\rangle + (\lambda y.\langle y + \mathcal{S}\,k_2.\mathcal{S}\,k_3.1\rangle)\ 10\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\,k.M]\rangle] \quad\rightsquigarrow\quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$$1$$

$$- + (\lambda y.\langle y + \mathcal{S}\,k_2.\mathcal{S}\,k_3.1\rangle)\ 10$$

$$1 + -$$

$$\langle 1 + \langle [\langle 1\rangle] + (\lambda y.\langle y + \mathcal{S}\,k_2.\mathcal{S}\,k_3.1\rangle)\ 10\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$$(\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 10$$

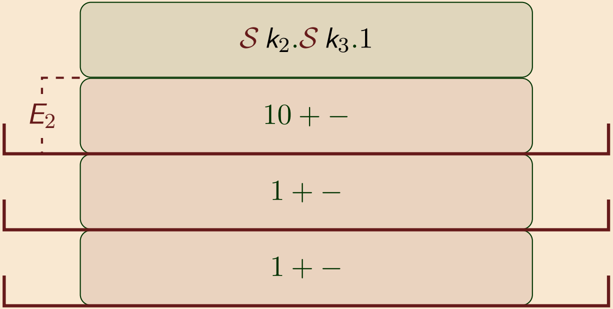
$$1 + -$$

$$1 + -$$

$$\langle 1 + \langle 1 + [(\lambda y.\langle y + \mathcal{S}\, k_2.\mathcal{S}\, k_3.1\rangle)\ 10]\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$



$$\langle 1 + \langle 1 + \langle 10 + [\mathcal{S}\, k_2.\mathcal{S}\, k_3.1]\rangle\rangle\rangle$$

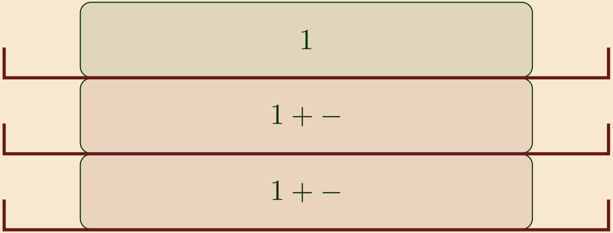
Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$$\langle 1 + \langle 1 + \langle [\mathcal{S}\, k_3.1]\rangle\rangle\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$



$$\langle 1 + \langle 1 + [\langle 1\rangle]\rangle\rangle$$

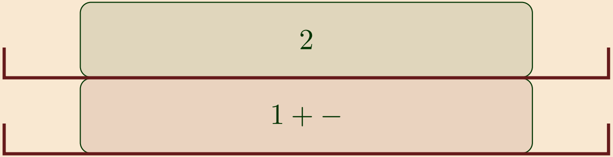
Example

$$E[\langle E_2[\mathcal{S}\,k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$$\langle 1 + \langle [1 + 1] \rangle \rangle$$

Example

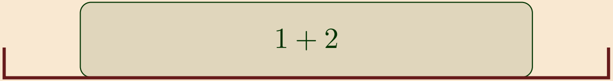
$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$



$$\langle 1 + [\langle 2\rangle]\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$



$$\langle[1+2]\rangle$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

$$[\langle 3\rangle]$$

Example

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \quad \rightsquigarrow \quad E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

3

3

Variations & applications

Variations

retain enclosing $\langle - \rangle$

install $\langle - \rangle$ around E_2

shift

$$E[\langle E_2[\mathcal{S}\, k.M]\rangle] \rightsquigarrow E[\langle M\{(\lambda y.\langle E_2[y]\rangle)/k\}\rangle]$$

shift0

$$E[\langle E_2[\mathcal{S}_0\, k.M]\rangle] \rightsquigarrow E[M\{(\lambda y.\langle E_2[y]\rangle)/k\}]$$

control

$$E[\langle E_2[\mathcal{F}\, k.M]\rangle] \rightsquigarrow E[\langle M\{(\lambda y.E_2[y])/k\}\rangle]$$

control0

$$E[\langle E_2[\mathcal{F}_0\, k.M]\rangle] \rightsquigarrow E[M\{(\lambda y.E_2[y])/k\}]$$

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\begin{array}{l}\text{try } b\ h = \text{case } \langle R\,(b\,())\rangle \text{ of } L\ e \to h\ e \mid R\ v \to v \\ \text{raise } e = \mathcal{S}_0\, k.L\ e\end{array}$$

Example:

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\begin{array}{l} \text{try } b\ h = \text{case } \langle R\,(b\,())\rangle \text{ of } L\ e \rightarrow h\ e \mid R\ v \rightarrow v \\ \text{raise } e = \mathcal{S}_0\, k.L\ e \end{array}$$

Example: $\text{try } (\lambda().1 + (\text{raise } 0 + 100))(\lambda u.u + 2)$

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\text{try } b\ h = \text{case } \langle R\ (b\ ())\rangle \text{ of } L\ e \rightarrow h\ e \mid R\ v \rightarrow v$$
$$\text{raise } e = \mathcal{S}_0\ k.L\ e$$

Example: $\text{try } (\lambda().1 + (\text{raise } 0 + 100))(\lambda u.u + 2)$

$\rightsquigarrow \text{case } \langle R((\lambda().1 + ([\mathcal{S}_0\ k.L\ 0] + 100))())\rangle \text{ of } L\ e \rightarrow (\lambda u.u + 2)\ e \mid R\ v \rightarrow v$

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\text{try } b\ h = \text{case } \langle R\,(b\,())\rangle \text{ of } L\ e \rightarrow h\ e \mid R\ v \rightarrow v$$
$$\text{raise } e = \mathcal{S}_0\, k.L\ e$$

Example: $\text{try } (\lambda().1 + (\text{raise } 0 + 100))(\lambda u.u + 2)$

$\rightsquigarrow \text{case } \langle R((\lambda().1 + ([\mathcal{S}_0\, k.L\ 0] + 100))()) \rangle \text{ of } L\ e \rightarrow (\lambda u.u + 2)\ e \mid R\ v \rightarrow v$

$\rightsquigarrow \text{case } \langle R(1 + [\mathcal{S}_0\, k.L\ 0])\rangle ofL\ e \rightarrow (\lambda u.u + 2)\ e \mid R\ v \rightarrow v$

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\text{try } b\ h = \text{case } \langle R\,(b\,())\rangle \text{ of } L\ e \to h\ e \mid R\ v \to v$$
$$\text{raise } e = \mathcal{S}_0\, k.L\ e$$

Example: $\text{try } (\lambda().1 + (\text{raise } 0 + 100))(\lambda u.u + 2)$

$\rightsquigarrow \text{case } \langle R((\lambda().1 + ([\mathcal{S}_0\ k.L\ 0] + 100))())\rangle \text{ of } L\ e \to (\lambda u.u + 2)\ e \mid R\ v \to v$

$\rightsquigarrow \text{case } \langle R(1 + [\mathcal{S}_0\ k.L\ 0])\rangle ofL\ e \to (\lambda u.u + 2)\ e \mid R\ v \to v$

$\rightsquigarrow [\text{case } L\ 0 \text{ of } L\ e \to (\lambda u.u + 2)e \mid R\ v \to v]$

The stack
Continuations
Variations & applications
Reading

Simulating exceptions is straightforward: just discard the continuation:

$$\text{try}\ b\ h = \text{case}\ \langle R\ (b\ ())\rangle\ \text{of}\ L\ e \rightarrow h\ e \mid R\ v \rightarrow v$$
$$\text{raise}\ e = \mathcal{S}_0\ k.L\ e$$

Example:

$$\text{try}\ (\lambda().1 + (\text{raise}\ 0 + 100))(\lambda u.u + 2)$$
$$\rightsquigarrow \text{case}\ \langle R((\lambda().1 + ([\mathcal{S}_0\ k.L\ 0] + 100))())\rangle\ \text{of}\ L\ e \rightarrow (\lambda u.u + 2)\ e \mid R\ v \rightarrow v$$
$$\rightsquigarrow \text{case}\ \langle R(1 + [\mathcal{S}_0\ k.L\ 0])\rangle ofL\ e \rightarrow (\lambda u.u + 2)\ e \mid R\ v \rightarrow v$$
$$\rightsquigarrow [\text{case}\ L\ 0\ \text{of}\ L\ e \rightarrow (\lambda u.u + 2)e \mid R\ v \rightarrow v]$$
$$\rightsquigarrow [(\lambda u.u + 2)\ 0]$$

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\text{try } b\ h = \text{case } \langle R\,(b\,())\rangle \text{ of } L\ e \to h\ e \mid R\ v \to v$$
$$\text{raise } e = \mathcal{S}_0\, k.L\ e$$

Example:

$$\text{try } (\lambda().1 + (\text{raise } 0 + 100))(\lambda u.u + 2)$$
$$\rightsquigarrow \text{case } \langle R((\lambda().1 + ([\mathcal{S}_0\, k.L\ 0] + 100))())\rangle \text{ of } L\ e \to (\lambda u.u + 2)\ e \mid R\ v \to v$$
$$\rightsquigarrow \text{case } \langle R(1 + [\mathcal{S}_0\, k.L\ 0])\rangle ofL\ e \to (\lambda u.u + 2)\ e \mid R\ v \to v$$
$$\rightsquigarrow [\text{case } L\ 0 \text{ of } L\ e \to (\lambda u.u + 2)e \mid R\ v \to v]$$
$$\rightsquigarrow [(\lambda u.u + 2)\ 0]$$
$$\rightsquigarrow [0 + 2]$$

Application: exceptions

Simulating exceptions is straightforward: just discard the continuation:

$$\text{try } b\ h = \text{case } \langle R\,(b\,())\rangle \text{ of } L\ e \to h\ e \mid R\ v \to v$$
$$\text{raise } e = \mathcal{S}_0\, k.L\ e$$

Example:

$$\text{try } (\lambda().1 + (\text{raise } 0 + 100))(\lambda u.u + 2)$$
$$\rightsquigarrow \text{case } \langle R((\lambda().1 + ([\mathcal{S}_0\ k.L\ 0] + 100))())\rangle \text{ of } L\ e \to (\lambda u.u + 2)\ e \mid R\ v \to v$$
$$\rightsquigarrow \text{case } \langle R(1 + [\mathcal{S}_0\ k.L\ 0])\rangle ofL\ e \to (\lambda u.u + 2)\ e \mid R\ v \to v$$
$$\rightsquigarrow [\text{case } L\ 0 \text{ of } L\ e \to (\lambda u.u + 2) e \mid R\ v \to v]$$
$$\rightsquigarrow [(\lambda u.u + 2)\ 0]$$
$$\rightsquigarrow [0 + 2]$$
$$\rightsquigarrow [2]$$

Application: generators

We can build generators that yield items from iterators that traverse collections

$$\text{generate iter } l = \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\, l \rangle$$

Example:

Application: generators

We can build generators that yield items from iterators that traverse collections

$$\text{generate iter } l = \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\, l \rangle$$

Example:

$$\text{generate iter } [1;2;3]$$

Application: generators

We can build generators that yield items from iterators that traverse collections

$$\text{generate iter } l = \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\, l \rangle$$

Example:

$\text{generate iter } [1;2;3]$

$\rightsquigarrow \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [1;2;3] \rangle$

Application: generators

We can build generators that yield items from iterators that traverse collections

$$\text{generate iter } l = \langle \text{iter } (\lambda v. \mathcal{S}_0\, k.(v, k))\, l \rangle$$

Example:

$$\text{generate iter } [1; 2; 3]$$
$$\rightsquigarrow \langle \text{iter } (\lambda v. \mathcal{S}_0\, k.(v, k))\ [1; 2; 3] \rangle$$
$$\rightsquigarrow \langle (\lambda v. \mathcal{S}_0\, k.(v, k))\ 1;\ \text{iter } (\lambda v. \mathcal{S}_0\, k.(v, k))\ [2; 3] \rangle$$

Application: generators

We can build generators that yield items from iterators that traverse collections

$$\text{generate iter } l = \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\, l \rangle$$

Example:

$$\text{generate iter } [1;2;3]$$
$$\rightsquigarrow \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [1;2;3] \rangle$$
$$\rightsquigarrow \langle (\lambda v.\mathcal{S}_0\, k.(v, k))\ 1;\ \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [2;3] \rangle$$
$$\rightsquigarrow \langle (\mathcal{S}_0\, k.(1, k));\ \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [2;3] \rangle$$

We can build generators that yield items from iterators that traverse collections

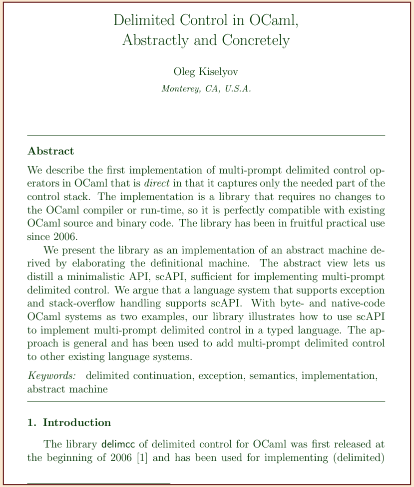
$$\text{generate iter } l = \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\, l \rangle$$

Example:

$$
\begin{aligned}
&\text{generate iter } [1;2;3] \\
&\rightsquigarrow \langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [1;2;3] \rangle \\
&\rightsquigarrow \langle (\lambda v.\mathcal{S}_0\, k.(v, k))\ 1;\ \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [2;3] \rangle \\
&\rightsquigarrow \langle (\mathcal{S}_0\, k.(1, k));\ \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [2;3] \rangle \\
&\rightsquigarrow (1, \lambda().\langle \text{iter } (\lambda v.\mathcal{S}_0\, k.(v, k))\ [2;3] \rangle)
\end{aligned}
$$

Reading

Paper 1: delimcc (2012)



Delimited Control in OCaml,
Abstractly and Concretely

Oleg Kiselyov
Monterey, CA, U.S.A.

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 run-time, 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 distill 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 of delimited control for OCaml was first released at the beginning of 2006 [1] and has been used for implementing (delimited)

"[T]he 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.

"[D]oes 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 re-instated arbitrarily many times in different dynamic contexts."

Paper 2: a selective CPS transform (2009)

Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform

Tiark Rompf   Ingo Maier   Martin Odersky

Programming Methods Laboratory (LAMP)
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{firstname.lastname}@epfl.ch

Abstract

We describe the implementation of first-class polymorphic delimited continuations in the programming language Scala. We use Scala's pluggable typing architecture to implement a simple type and effect system, which discriminates expressions with control effects from those without and accurately tracks answer type modification incurred by control effects. To 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. Benchmarks indicate that this high-level approach performs competitively.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Languages, Theory

Keywords Delimited continuations, selective CPS transform, control effects, program transformation

1. Introduction

Continuations, and in particular delimited continuations, are a versatile programming tool. Most notably, we are interested in their ability to suspend and resume sequential code paths in a controlled way without syntactic overhead and without being tied to VM threads.

Classical (or *full*) continuations can be seen as a functional version of the infamous GOTO-statement (Strachey and Wadsworth 2000). Delimited (or *partial*, or *composable*) continuations are more like regular functions and less like GOTOs. They do not embody the entire rest of the computation, but just a partial rest, up to a programmer-defined outer bound. Unlike their undelimited counterparts, delimited continuations will actually return control to the caller after they are invoked, and they may also return values. This means that delimited continuations can be called multiple times in succession, and the program can proceed at the call site afterwards. This ability makes delimited continuations strictly more powerful than regular ones. Operationally speaking, delimited continuations do not embody the entire control stack but just stack fragments, so they can be used to recombine stack fragments in interesting and possibly complicated ways.

To access and manipulate delimited continuations in direct-style programs, a number of control operators have been proposed, which can be broadly classified as static or dynamic, according to whether the extent of the continuations they capture is determined statically or not. The dynamic variant is due to Felleisen (1988); Felleisen et al. (1988) and the static variant to Danvy and Filinski (1990, 1992). The static variant has a direct, corresponding CPS-formulation which makes it attractive for an implementation using a static code transformation and thus, this is the variant underlying the implementation described in this paper. We will not go into the details of other variants here, but refer to the literature instead (Dyvbig et al. 2007; Shan 2004; Biernacki et al. 2006); suffice it to note that the two main variants, at least in an untyped setting, are equally expressive and have been shown to be macro-expressible (Felleisen 1991) by each other (Shan 2004; Kiselyov 2005). Applying the type systems of Asai and Kameyama (2007); Kameyama and Yonezawa (2008), however, renders the dynamic control operators strictly more expressive since strong normalization holds only for the static variant (Kameyama and Yonezawa 2008).

In Danvy and Filinski's model, there are two primitive operations, `shift` and `reset`. With `shift`, one can access the current continuation and with `reset`, one can demarcate the boundary up to which continuations reach: A `shift` will capture the control context up to, but not including, the nearest dynamically enclosing `reset` (Biernacki et al. 2006; Shan 2007).

Despite their undisputed expressive power, continuations (and in particular delimited ones) have not yet found their way into the majority of programming languages. Full continuations are standard language constructs in Scheme and popular ML dialects, but most other languages do not support them natively. This is partly because efficient support for continuations is assumed to require 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 simulate stack inspection with exception handlers and/or external data structures (Pettyjohn et al. 2005; Srinivasan 2006).

Another avenue is to use monads instead of continuations to express custom-defined control flow. Syntactic restrictions imposed by monadic style can be overcome by supporting more language constructs in the monadic level, as is done in F#'s workflow expressions. Nevertheless, the fact remains that monads or workflows impose a certain duplication of syntax constructs that need to be

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

"To 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.

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

Paper 3: WasmFX (2023)

Continuing WebAssembly with Effect Handlers

LUNA PHIPPS-COSTIN, Northeastern University, United States
ANDREAS ROSSBERG, Independent, Germany
ARJUN GUHA, Northeastern University and Roblox, United States
DAAN LEIJEN, Microsoft Research, United States
DANIEL HILLERSTRÖM, Huawei Zurich Research Center, Switzerland
KC SIVARAMAKRISHNAN, Tarides and IIT Madras, India
MATIJA PRETNAR, University of Ljubljana and Institute of Mathematics, Physics & Mechanics, Slovenia
SAM LINDLEY, The University of Edinburgh, United Kingdom

WebAssembly (Wasm) is a low-level portable code format offering near native performance. It is intended as a compilation target for a wide variety of source languages. However, Wasm provides no direct support for non-local control flow features such as async/await, generators/iterators, lightweight threads, first-class continuations, etc. This means that compilers for source languages with such features must ceremoniously transform whole source programs in order to target Wasm.

We present *WasmFX*, an extension to Wasm which provides a universal target for non-local control features via *effect handlers*, enabling compilers to translate such features directly into Wasm. Our extension is minimal and only adds three main instructions for creating, suspending, and resuming continuations. Moreover, our primitive instructions are type-safe providing typed continuations which are well-aligned with the design principles of Wasm whose stacks are typed. We present a formal specification of WasmFX and show that the extension is sound. We have implemented WasmFX as an extension to the Wasm reference interpreter and also built a prototype WasmFX extension for Wasmtime, a production-grade Wasm engine, piggybacking on Wasmtime's existing fibers API. The preliminary performance results for our prototype are encouraging, and we outline future plans to realise a native implementation.

CCS Concepts: • **Theory of computation → Control primitives**; **Operational semantics**.

Additional Key Words and Phrases: WebAssembly, effect handlers, stack switching

ACM Reference Format:
Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 238 (October 2023), 27 pages. https://doi.org/10.1145/3622814

1 INTRODUCTION

WebAssembly (also known as Wasm) [Haas et al. 2017; Rossberg 2019, 2023] is a low-level virtual machine designed to be safe and fast, while being both language- and platform-independent. A

"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?