Delimited continuations

1 Readings

1.1 Set papers

The week’s set papers are as follows:

- Delimited control in OCaml, abstractly and concretely (Kiselyov, 2012)
- Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform (Rompf, Maier, and Odersky, 2009)
- Continuing WebAssembly with Effect Handlers (Phipps-Costin, Rossberg, Guha, et al., 2023)

You are invited to write an essay about two of these papers, following the guidelines on the course assessment page.

1.2 Background

Appendix A of Foundations for Programming and Implementing Effect Handlers (Hillerström, 2021) is a recent overview of delimited continuations, with details on the history, on the various operators with their semantics and types, on applications, and on implementation approaches.

2 History

2.1 Beginnings

Continuations were used to define the semantics of jumps in programming languages from the early years of computer science (The Discoveries of Continuations (Reynolds, 1993) lists some examples in the early 1960s). Undelimited continuations as a programming construct were introduced in Defini

2.2 Developments

1987 Beyond Continuations (Felleisen, Friedman, Duba, et al., 1987) and Abstract Continuations: A Mathematical Semantics for Handling Full Jumps (Felleisen, Wand, Friedman, et al., 1988) noted some complications with the axioms for undelimited continuations, and introduced a delimited form, $\mathcal{F}$ (control).

1989 A Functional Abstraction of Typed Contexts (Danvy and Filinski, 1989) introduced an alternative operator for capturing delimited continuations, $\text{shift}$, noting that it could be statically typed.

1995 A Generalization of Exceptions and Control in ML-like Languages (Gunter, Rémy, and Riecke, 1995) introduced yet another delimited control operator, $\text{cupto}$, integrated into an ML-style language and type system.

2007 Polymorphic Delimited Continuations (Asai and Kameyama, 2007) introduced a type system for a language with delimited continuations that distinguishes between effectful expressions (that use $\text{shift}$ and $\text{reset}$) and pure expressions (that do not).

2.3 Status

Delimited continuations have proved to be a very general mechanism for implementing effects of various types. (Representing Monads (Filinski, 1994) shows that they can be used to embed any expressible monad.) They have been usefully employed, for example, to simulate exceptions, nondeterminism, mutable state, concurrency, generators, and in more specialized applications in probabilistic programming, automatic differentiation, partial evaluation, and linguistics.

However, despite these many applications and despite the long history of study, relatively few general-purpose languages provide built-in support for delimited continuations. OCaml 5 is one exception, and there is a proposal under consideration to add support for delimited continuations to WebAssembly (Phipps-Costin et al., 2023), where it may be used as a uniform basis for a variety of non-local control features including threads and async/await.

1 https://www.cl.cam.ac.uk/teaching/2324/R277/assessment.html
We start by showing how delimited continuations work in concert: \( S \cdot k.M \) captures the context up to the nearest enclosing \( (\cdot) \) and inserts it into \( M \) in place of the bound variable \( k \). Variants of these rules are given by many authors, including Kiselyov, Shan, and Sabry (2006) and Ishio and Asai (2023).

### 3.1 Operator taxonomy

There are three common alternatives to \( \text{shift} \) (Figure 2), which vary (on the right of the reduction rule) as to whether they retain the \( \text{reset} \) in the current context and whether they enclose the captured context. Ishio and Asai (2023) and many others give details.

### 3.2 Examples

We start by showing how delimited continuations can implement exceptions. The expression \( \text{try } b \ h \) first evaluates \( b \) \( () \); if that concludes without invoking a control operator, its value becomes the value of the whole expression. However, \( b \) \( () \) might also call \( \text{raise} \) \( e \), discarding the continuation \( k \) up to the call to \( b \) and passing \( e \) to the handler function \( h \):

\[
\text{try } b \ h = \text{case } \langle R \ (b \ () ) \rangle \text{ of } L \ e \to h \ e \ | \ R \ v \to v \ \text{raise } e = \mathcal{S}_0 \ k \ L \ e
\]

We next show an example that turns an arbitrary iterator (e.g. OCaml’s \texttt{List.iter}) into a generator like Python’s \texttt{yield}:

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

Here \texttt{generate iter} \( l \) installs a reset and calls the function \texttt{iter}, passing it a function that captures the continuation \( k \) up to the reset, passing both the \( k \) and the function argument \( v \). The caller can process the value \( v \) and re-invoke the continuation \( k \) continuation to resume the iteration.

Finally, we show an example that distinguishes various control operators:

\[
\langle 1 + \langle S \ k_1. k_1 100 + k_1 10 + S \ k_2. S(v \ k_3.1) \rangle \rangle
\]

As written, this example reduces to \( 3 \); however, it can be modified to instead reduce to \( 1 \) or \( 2 \) by respectively changing \( S \) to \( S_0 \) or to \( \mathcal{F} \) throughout.
4 Implementations and applications

4.1 Implementation considerations

4.1.1 Types

Type systems for delimited continuations have been studied since at least Danvy and Filinski (1989), but remain challenging due to answer-type modification, i.e. changes to the return type of continuations arising from use of \texttt{shift}. Kobori, Kameyama, and Kiselyov (2015) give details and a type-preserving translation that eliminates answer-type modification.

4.1.2 Named prompts

Since delimited control operators like \texttt{shift} capture continuations up to the nearest enclosing \langle – \rangle, multiple uses of \texttt{shift} within a program can sometimes interact unexpectedly. To help avoid such interactions, some implementations support named reset points that allow \texttt{shift} to indicate explicitly how much of the enclosing continuation to capture.

4.1.3 Multi-shot vs one-shot

Implementations of delimited continuations vary in whether they allow captured continuations to be invoked multiple times (so-called multi-shot continuations), or at most once (single-shot).

4.2 Implementations

There are a variety of approaches to implementing delimited continuations, many of which involve changing compilation (e.g. to transform programs into continuation-passing style), modifying the language runtime (e.g. to provide primitives for manipulating stacks), or both. Other implementation strategies rely on the expressibility of delimited continuations in terms of other language constructs such as threads or undelimited continuations.

- **Threads Yield Continuations** (Kumar, Brugeman, and Dybvig, 1998) implements one-shot and multi-shot delimited continuations using threads. (In the other direction, delimited continuations can implement cooperative threads (Dolan et al., 2015).)

- **Representing Monads** (Filinski, 1994) shows how to implement \texttt{shift} and \texttt{reset} using undelimited continuations and mutable reference cells.

- **Final shift for call/cc: direct implementation of \texttt{shift} and \texttt{reset}** (Gasbiblicher and Sperber, 2002) gives a direct Scheme48 implementation of \texttt{shift} and \texttt{reset}, and shows improved efficiency over a simulation using undelimited continuations.

- **Continuations from generalized stack inspection** (Pettijohn, Clements, Marshall, et al., 2005) uses exceptions and code that saves and restores local data to simulate delimited control

- **Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform** (Rompf, Maier, and Odersky, 2009) uses the type system of Asai and Kameyama (2007) to CPS-convert only those parts of a program that actually make use of control effects.

- **Delimited control in OCaml, abstractly and concretely** (Kiselyov, 2012) implements delimited continuations as an OCaml library using low-level stack-manipulation primitives, without modifying the language implementation.

4.3 Applications

Support for interactions between programs and their contexts has many applications.

In partial evaluation and its more explicit variant multi-stage programming, delimited continuations can be used to rearrange program fragments so that they specialize more effectively (Lawall and Danvy, 1994; Kameyama, Kiselyov, and Shan, 2011).

In linguistics, lambda-calculus based formalisms such as Combinatory Categorical Grammar extended with delimited continuations can capture interactions between linguistic expressions and scope-sensitive phenomena (e.g. quantifiers) (Shan, 2004).

Reverse-mode automatic differentiation can be implemented more straightforwardly using delimited continuations than with traditional methods such as tapes (Wang et al., 2019).

Delimited continuations have also been used to effectively embed domain-specific languages for probabilistic programming in general purpose host languages (Kiselyov and Shan, 2009).

4.4 Connection to effect handlers

Effect handlers are a recent approach to supporting user-defined effects. They are close in expressive power to delimited continuations, but have a more convenient programming interface that resembles resumable exceptions. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control (Forster, Kammar, Lindley, et al., 2017) gives a careful expressivity comparison; Handlers in action (Kammar, Lindley, and Oury, 2013) shows how delimited continuations can be used to implement effect handlers.
References


