Delimited Control - Presentation Notes

Stephen Chang

call/cc

call/cc. Who here has seen it before? Who here is comfortable programming with it? Who here likes programming with it? Luckily for all of you, this presentation is not about call/cc. But it is relevant to my topic so I'm going to start with a review of call/cc.

- show semantics
- show small examples

As I mentioned, this talk is not about the merits of call/cc, so in this talk, I'm going to assume that call/cc is accepted as something that's useful to have in a programming language. call/cc is useful because it is a very general control operator that can be used to implement a variety of other control operators. What can you do with it?

- show exception handling example

For more complex call/cc examples, we can look at the paper Constraining Control, by Friedman and Haynes. In this paper Friedman and Haynes give several use cases for call/cc in which call/cc either needs to be extended or constrained.

Constraining Control

The paper starts with an implementation of a fluid environment. A fluid environment is used to achieve dynamic binding.

- show fluid-let example

In a language, with call/cc, this implementation may not work all the time, for example, if a piece of code in the body jumps out of the body and into another fluid-let body, the proper environment will not be used. To fix this problem, we need to extend call/cc.

- show call/cc-fluid example

What we're doing is saving the fluid environment at the time the continuation was captured, and then installing this environment every time the captured continuation is invoked. We must extend the captured continuation to perform this installation and we achieve this by extending call/cc.

Now let's look at another example

– show call/cc-oneshot

From this example and the previous one, we can see a general pattern forming. Essentially, the captured continuation is itself captured in a lambda, where other operations are performed before the continuation is invoked.

In this example, we are constraining the continuation captured by call/cc in that we only want to allow the captured continuation to be invoked once. A real world situation where this situation is applicable could be bank software where the captured continuations represent queued up banking transactions. However, there is a problem with this code. The problem is that if there is another continuation that overlaps with this one, then it's still possible for the code in this continuation to be executed more than once. For example, if you have a program E[M] where E = E1[E2[]], and both E and E1 are captured by calls to callcc, even though they are both "one-shot", E1 will still be executed twice.

So now things are getting complicated. To fix this situation, the authors link related continuations so that when one is invoked, it can disable all continuations that are related. To do this, they need to implement a way to link the continuations, and also a protocol for continuations to send messages or commands to each other.

At this point I'm going to conclude wrap up the presentation of this paper. What I wanted to demonstrate was that call/cc is useful but must be properly harnessed and doing so could be complicated. And we havent even tried to combine all these call/cc extensions. To wrap up, here are some more tools that the authors implement:

– continuation linkages
– continuation messaging
– dynamic domains

Ultimately the authors utilize all these tools to implement an unwind-protect mechanism: (unwind-protect ¡body¿ ¡postlude¿). unwind-protect guarantees that the code in the postlude will always be executed whenever either the body completes executing, or control leaves the body. It's useful for implementing cleanup, like closing files, or to perform other operations that ensure the system is in a stable state. The authors also generalize unwind-protect to dynamic-wind, which accepts an additional prelude parameter. The only reason for mentioning this is that we'll see it again later in the talk.

Last observation, it seems that most non-trivial uses of call/cc require a heap (ie - set!).

takeaways:

– call/cc is useful but complicated to use
– call/cc usually requires a heap to use
– unwind-protect

The Theory and Practice of First-Class Prompts

Since call/cc is difficult to use, some enterprising researchers thought, hey, maybe we should try to formalize this beast. And that brings us to my second paper. After working for a while, Felleisen came up with the λ_v–C calculus, which is an extension of the call-by-value lambda calculus that includes a call/cc-like operator F. Felleisen used F instead of call/cc because it made the calculus nicer (less reduction rules).

- introduce calculus
- compare F to call/cc
- show call/cc implemented with F

The problem with the calculus is that you can no longer reason locally about terms. For example, $\mathcal{F}\ \lambda k.0$ and 0 reduce to the same value, but not when placed in a non-empty context. To fix this, Felleisen added a prompt operator. The prompt delimits the continuation captured by $\mathcal{F}$.

- introduce prompt extension to calculus and reduction rules
- introduce abstract machine
- show examples
- show benefit of not needing heap

Summary

Prompts introduced to address problem with Felleisen's λ_v–C calculus [1]

λ_v–C calculus

1. λ_v–C calculus terms:

$$M ::= c \mid x \mid \lambda x.M \mid M\ N \mid \mathcal{F}\ M$$

   $\mathcal{F}$ operator is like call/cc - evaluates argument and then applies it to current continuation. The key difference is that $\mathcal{F}$ itself aborts the current continuation, instead of the current continuation itself being abortive. Therefore:

$$(\texttt{add1}\ (\mathcal{F}\ (\lambda k.(k\ (k\ 1))))) = 3$$

   but

$$(\texttt{add1}\ (\texttt{call/cc}\ (\lambda k.(k\ (k\ 1))))) = 2$$

   and

$$(\texttt{add1}\ (\mathcal{F}\ (\lambda k.0))) = 0$$

   but

$$(\texttt{add1}\ (\texttt{call/cc}\ (\lambda k.0))) = 1$$

2. Notions of reduction for $\mathcal{F}$:

$$\mathcal{F}_L : (\mathcal{F}M)N \to \mathcal{F}(\lambda k.M(\lambda m.k(mN))))$$

$$\mathcal{F}_R : V(\mathcal{F}M) \to \mathcal{F}(\lambda K.M(\lambda m.k(Vm))))$$

"Purpose of these reductions is to push $\mathcal{F}$ to the root of a term and to encode the context of the $\mathcal{F}$ application as an abstraction. Essentially, the rules are building up the abstraction that represents the continuation one frame at a time.

3. Once the $\mathcal{F}$ reaches the root, then we can get rid of it with the rule:

$$\mathcal{F}M \triangleright M(\lambda x.x)$$

What the rule is saying that the last context is just the identify context (a hole).

4. However, the $\mathcal{F}$ elimination rule only applies when $\mathcal{F}$ is at the root and thus causes problems when trying to prove operational equivalence of the calculus because two terms that evaluate to the same value can behave differently when placed in the same context. Example: $\mathcal{F}(\lambda d.0)$ vs 0. First term aborts current context and evaluates to 0 while second one just evaluates to 0 in the current context.
5. To fix the operational equivalent problem, a "prompt" operator # is added to the calculus, along with an accompanying reduction rule:

$$M ::= c \mid x \mid \lambda x.M \mid MN \mid \mathcal{F}M \mid \#M$$

$$\#_{\mathcal{F}} : (\#(\mathcal{F}M)) \to (\#(M(\lambda x.x)))$$

However, now prompts are first class. They can appear anything in a term. This will prove to be quite useful. This is a key difference compared to the Haynes-Friedman method of constraining control.

6. CEK machine transistions for $\mathcal{F}$ and #:

$$\langle\ \#M,\ \rho,\ \kappa\ \rangle \overset{CEK}{\longmapsto} \langle\ M,\ \rho,\ (\kappa\ \mathbf{mark})\ \rangle$$

$$\langle\ \mathcal{F}M,\ \rho,\ \kappa\ \rangle \overset{CEK}{\longmapsto} \langle\ M\gamma,\ \rho[\gamma := \langle\ \mathbf{p},\ \oplus\kappa\ \rangle],\ \ominus\kappa\ \rangle$$

$$\langle\ \ddagger,\ \emptyset,\ ((\kappa\ \mathbf{fun}\ \langle\ \mathbf{p},\ \kappa_0\ \rangle)\ \mathbf{ret}\ V)\ \rangle \overset{CEK}{\longmapsto} \langle\ \ddagger,\ \emptyset,\ (\kappa \otimes \kappa_0\ \mathbf{ret}\ V)\ \rangle$$

Boldface words are tags for a continuation. Prompt marks the stack. Mark is removed after sub-evaluation terminates $\oplus$ operator copies the stack up to the next mark. $\ominus$ operator deletes the stack up the next mark. $\otimes$ is stack append operator. **ret** continuation means subterm is done evaluating and value has been reached.

Prompt Usage Examples

1. closing files: $\lambda p\ \ f_{open}.$(**begin**(#($p\ \ f_{open}$))(close f_{open})) above example is same as unwind-protect, but less complicated
2. interpreter:

```
LOOP == (iterate L (exp (prompt-read ' ->))
         (if (eq? exp 'exit) 'good-bye
             (begin
               (# (evaluate exp base-environment))
               (L (prompt-read ' ->)))))
```

   If object language imports something like call/cc from implementation language, then entire interpreter could be aborted if call/cc appears in `exp`

Control Delimiters and Their Hierarchies

```
@article{Sitaram1990Hierarchies,
  author    = {Dorai Sitaram and Matthias Felleisen},
  title     = {Control Delimiters and Their Hierarchies},
  journal   = {Lisp and Symbolic Computation},
  volume    = {3},
  number    = {1},
  year      = {1990},
  pages     = {67-99},
}
```

Abstract

Since control operators for the *unrestricted* transfer of control are too powerful in many situations, we propose the *control delimiter* as a means for restricting control manipulations and study its use in Lisp- and Scheme-like languages. In a Common Lisp-like setting, the concept of delimiting control provides a well-suited terminology for explaining different control constructs. For higher-order languages like Scheme, the control delimiter is the means for embedding Lisp control constructs faithfully and for realizing high-level control abstractions elegantly. A deeper analysis of the examples suggests a need for an entire *control hierarchy* of such delimiters. We show how to implement such a hierarchy on top of the simple version of a control delimiter.

`control` and `run`

1. `control` operator in this paper = $\mathcal{F}$ operator from Felleisen 1998 (captures and uses non-abortive continuations, operator itself aborts the current continuation)
2. `call/cc` implemented using `control`:

```
(define (call/cc f)
  (control (lambda (k)
             (k (f (lambda (v) (abort (k v))))))))
```

where

$$(\texttt{abort}\ exp) \equiv (\texttt{control}\ (\lambda dummy.exp))$$

The outer application of the captured continuation k is needed because `control` aborts its current continuation while `call/cc` does not. The `abort` is needed because `call/cc` captures abortive continuations while `control` does not.

3. simulating `control` with `call/cc` is inefficient bc `control` is stack-based and `call/cc` is heap-based
4. Simulating `control` and `run` with `call/cc` would requiring repackaging of the entire control stack into just portions of the control stack. In addition, we would need to keep track of specific points on the stack that are delimited. So complicated that Sitaram and Felleisen say in their paper "skip this section on a first reading"

implementing control operators with `control` and `run`

1. `catch` and `throw`

$$(\texttt{throw}\ tag\ value) \equiv (\texttt{abort}\ (list\ 'throw\ tag\ value))$$

$$(\texttt{catch}\ tag\ exp) \equiv$$

```
(let ([catch-tag tag] [result (% exp)])
  (record-case result
    ['throw (throw-tag throw-value)
            (if (eq? throw-tag catch-tag) throw-value
                (throw throw-tag throw-value))]
    [else result]))])
```

2. **unwind-protect**
   naive implementation:

$$(\textbf{unwind-protect}\ body\ postlude) \equiv (\texttt{begin0}\ (\%\ body)\ postlude)$$

However, you may want to do things after the postlude, like when there's a throw/catch

hierarchies

1. "multiple uses of `control` and `run` have the potential of interfering with each other"

$$(\texttt{catch}\ 'k\ (list\ (\%\ (\texttt{add1}\ (\texttt{throw}\ 'k\ 6)))))$$

delimiter does not allow throw to reach catch

2. "most natural solution calls for matching pairs of `control` and `run` ... However, total independence between all pairs of `control` and `run` is not always desirable"
3. proposal is to create hierarchy where each `control` and `run` get a "level" and a "prompt serves as control delimiter to all `control` of and above its level"
4. having a hierarchy also nicely protects the language from the programmer - language can still provide generic control operators to programmer but those operators will not conflict with already built-in language control operators

Abstracting Control

```
@inproceedings{Danvy1990AbstractingControl,
  author    = {Olivier Danvy and Andrzej Filinski},
  title     = {Abstracting Control},
  booktitle = {LISP and Functional Programming},
  year      = {1990},
  pages     = {151-160},
}
```

Abstract

The last few years have seen a renewed interest in continuations for expressing advanced control structures in programming languages, and new models such as Abstract Continuations have been proposed to capture these dimensions. This article investigates an alternative formulation, exploiting the latent expressive power of the standard continuation-passing style (CPS) instead of introducing yet other new concepts. We build on a single foundation: abstracting control as a *hierarchy* of continuations, each one modeling a specific language feature as acting on nested *evaluation contexts*.

We show how *iterating* the continuation-passing conversion allows us to specify a wide range of control behavior. For example, two conversions yield an abstraction of Prolog-style backtracking. A number of other constructs can likewise be expressed in this framework; each is defined independently of the others, but all are arranged in a hierarchy making any interactions between them explicit.

This approach preserves all the traditional results about CPS, e.g., its evaluation order independence. Accordingly, our semantics is directly implementable in a call-by-value language such as Scheme or ML. Furthermore, because the control operators denote simple, typable lambda-terms in CPS, they themselves can be statically typed. Contrary to intuition, the iterated CPS transformation does not yield huge results: except where explicitly needed, all continuations beyond the first one disappear due to the extensionality principle (η-reduction).

Besides presenting a new motivation for control operators, this paper also describes an improved conversion into applicative-order CPS. The conversion operates in one pass by performing all administrative reductions at translation

time; interestingly, it can be expressed very concisely using the new control operators. The paper also presents some examples of nondeterministic programming in direct style.

Summary

1. iterating CPS transformations gives a natural hierarchy of delimited control operators - "term is evaluated in a collection of embedded contexts, each represented by a continuation ... essentially, this generalizes ordinary CPS to a hierarchy of continuations, one for each context"
2. since CPS is used, continuations are actually represented as functions (in Felleisen, they are only "concatenable sequences of activation frames")
3. "replace the fundamentally dynamic control scoping specified by prior definitions of composable continuations with a properly static approach"
4. standard CPS transform (overbar operator) (with shift and reset)

$$\begin{aligned}
\overline{x} &\overset{CPS}{\longmapsto} \lambda\kappa.\kappa\ x \\
\overline{\lambda x.e} &\overset{CPS}{\longmapsto} \lambda\kappa.\kappa\ (\lambda x.\overline{e}) \\
\overline{e_1\ e_2} &\overset{CPS}{\longmapsto} \lambda\kappa.\overline{e_1}\ (\lambda f.\overline{e_2}\ (\lambda v.f\ v\ \kappa)) \\
\overline{\xi k.e} &\overset{CPS}{\longmapsto} \lambda\kappa.\overline{e}\ (\lambda x.x)\ [k \leftarrow \lambda v\kappa'.\kappa'\ (\kappa\ v)] \\
\overline{\langle e \rangle} &\overset{CPS}{\longmapsto} \lambda\kappa.\kappa\ (\overline{e}\ (\lambda x.x))
\end{aligned}$$

5. The composition of continuations in the k of shift is what differentiates shift and control. In control dynamically manipulates continuation frames, and the the equivalent of the composition of continuations is the appending of (partial) stacks. Operationally:

$$\begin{aligned}
M[(\texttt{reset}\ C[(\texttt{shift}\ f\ e)])] &\triangleright M[(\texttt{reset}\ e[f \leftarrow \lambda x.(\texttt{reset}\ C[x])])] \\
M[(\texttt{reset}\ C[(\texttt{control}\ f\ e)])] &\triangleright M[(\texttt{reset}\ e[f \leftarrow \lambda x.C[x]])]
\end{aligned}$$

   So

```
(reset (let ((y (shift f (cons 'a (f empty)))))
         (shift g y))) --> '(a)
```

   but

```
(reset (let ((y (control f (cons 'a (f empty)))))
         (shift g y))) --> '()
```

6. In regular denotational semantics, the shift and reset operators are:

$$\begin{aligned}
\mathcal{E}[\![\xi k.e]\!]\rho\kappa &= \mathcal{E}[\![e]\!]\rho[[\![k]\!] \leftarrow \lambda v\kappa'.\kappa'\ (\kappa\ v)](\lambda x.x) \\
\mathcal{E}[\![\langle e \rangle]\!]\rho\kappa &= \kappa\ (\mathcal{E}[\![e]\!]\rho(\lambda x.x))
\end{aligned}$$

7. in the translation of shift, there is a composition of continuations, but this results in a non-tail call. to fix this, we can apply CPS transformation again:

$$\mathcal{E}[\![\xi k.e]\!]\rho\kappa\gamma = \mathcal{E}[\![e]\!]\rho[[\![k]\!] \leftarrow \lambda v\kappa'\gamma''.\kappa\ v\ (\lambda w.\kappa'\ w\ \gamma'')](\lambda x\gamma'.\gamma'\ x)\gamma$$
$$\mathcal{E}[\![\langle e\rangle]\!]\rho\kappa\gamma = \mathcal{E}[\![e]\!]\rho(\lambda x\gamma'.\gamma'\ x)(\lambda v.\kappa\ v\ \gamma)$$

References

1. Felleisen, M., Friedman, D. P., Kohlbecker, E. E., and Duba, B. F. A syntactic theory of sequential control. *Theor. Comput. Sci. 52* (1987), 205–237.