Delimited Control - Presentation Notes

Stephen Chang

call/cc

1. call/cc applies its argument, which should be a function, to the current continuation, captured as a function continuation. If the captured continuation is ever applied, then it aborts whatever the continuation is at that time and replaces it with the captured continuation.
2. $$E[\texttt{call/cc}\ f] \rightarrow E[f\ k]$$

   where $k = \lambda v.(\texttt{abort}\ E[v])$ is the current continuation E captured as a function and

   $$E[\texttt{abort}\ V] \rightarrow V$$
3. $$(\texttt{add1}\ (\texttt{call/cc}\ \lambda k.0)) = 1$$

   because the captured continuation was never applied so the current continuation is not aborted

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

   because as soon as the inner k is applied, the other application of k is aborted

History

non-delimited:

1. call/cc
2. Landin's J
3. Reynolds - escape

delimited:

1. LISP - catch and throw - not first class - can only throw once to each catch
2. LISP - errset (mentioned in Sitaram and Felleisen)
3. Stoy and Strachey's run (= prompt) (in their experimental OS OS6) (mentioned in Sitaram and Felleisen)

Constraining Control

```
@inproceedings{Friedman1985ConstrainingControl,
  author    = {Daniel P. Friedman and Christopher T. Haynes},
  title     = {Constraining Control},
  booktitle = {POPL},
  year      = {1985},
  pages     = {245-254},
}
@article{Haynes1987EmbeddingContinuations,
  author    = {Christopher T. Haynes and Daniel P. Friedman},
  title     = {Embedding Continuations in Procedural Objects},
  journal   = {ACM Trans. Program. Lang. Syst.},
  volume    = {9},
  number    = {4},
  year      = {1987},
  pages     = {582-598},
}
```

Abstract

Continuations, when available as first-class objects, provide a general control abstraction in programming languages. They liberate the programmer from specific control structures, increasing programming language extensibility. Such continuations may be extended by embedding them in functional objects. This technique is first used to restore a fluid environment when a continuation object is invoked. We then consider techniques for constraining the power of continuations in the interest of security and efficiency. Domain mechanisms, which create dynamic barriers for enclosing control, are implemented using fluids. Domains are then used to implement an unwind-protect facility in the presence of first-class continuations. Finally, we demonstrate two mechanisms, wind-unwind and dynamic-wind, that generalize unwind-protect.

Extending call/cc

1. extend call/cc - instead of passing current continuation to call/cc argument, pass closure that contains continuation - closure can also do other stuff

```
(define (call/cc-extended f)
  (call/cc
    (lambda (k)
      (let ([cob (lambda (v)
                   extra operations
                   (k v))])
        (f cob)))))
```

2. an extended call/cc is needed when implementing a fluid environment in a language with first class continuations
3. standard fluid env impl (== means syntactic extension):

   ```
   (fluid <var>) == (fluid-env '<var>)

   (let-fluid <var> <exp> <body>) ==

   (let ([own-env fluid-env] ; save previous fluid env
         [v <exp>]
         [body-thunk (lambda () <body>)]) ; don't want to eval body until new env is set
     (set! fluid-env ; extend env
       (lambda (x)
         (if (eq? x '<var>) v (own-env x))))
     (begin0
       (body-thunk)
       (set! fluid-env own-env)))
   ```

4. implementation above does not work with first class continuations - when continuation is invoked, it should continue in the same fluid env as when continuation was first captured
5. extended call/cc for correctly handling fluid envs:

   ```
   (define (call/cc-fluid f)
     (call/cc
       (lambda (k)
         (f (let ([own-env fluid-env]) ; capture env at the time continuation is created
              (lambda (x)
                (set! fluid-env own-env)
                (k v)))))))
   ```

   Why dont we need to restore old fluid-env when call/cc is done? Because call/cc aborts current continuation.

Constraining call/cc

1. sometimes we want to constrain continuations - example - continuation that can only be invoked once

   ```
   (define (call/cc-one-shot f)
     (call/cc
       (lambda (k)
         (f (let ([alive #t])
              (lambda (v)
                (if alive
                    (begin
                      (set! alive #f)
                      (k v)
                    (error ...))))))))
   ```

Of course, this example is not completely robust in that descendants of the continuation can still be invoked even if alive is false. To address this, we need a more complicated scheme where each continuation needs to keep track of all child continuations and be able to set all their alive flags to false. Sitaram and Felleisen call this solution a "complicated communication system". See paper for more info. For now we assume that this is not an issue.

2. another example of constrained continuation - may only want to allow continuation jump if we are in a certain context
3. can implement using fluid envs:

```
(define (domain thunk)
  (let-fluid domain-ref (unique) (thunk)))

(define (call/cc-domain f)
  (call/cc-fluid
    (lambda (k)
      (f (let ([own-d (fluid domain-ref)]) ; save domain at cont creation
           (lambda (v)
             (if (eq? (fluid domain-ref own-d) ; check that we are in proper domain
                 (k v)
                 (error ...)))))))))
```

The Theory and Practice of First-Class Prompts

```
@inproceedings{Felleisen1998Prompts,
  author    = {Matthias Felleisen},
  title     = {The Theory and Practice of First-Class Prompts},
  booktitle = {POPL},
  year      = {1988},
  pages     = {180-190},
}
```

Abstract

An analysis of the λ_v-C-calculus and its problematic relationship to operational equivalence leads to a new control facility: the prompt-application. With the introduction of prompt-applications, the control calculus becomes a traditional calculus all of whose equations imply operational equivalence. In addition, prompt-applications enhance the expressiveness and efficiency of the language. We illustrate the latter claim with examples from such distinct areas as systems programming and tree processing.

Summary

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

$\boldsymbol{\lambda_v}$–$\boldsymbol{C}$ 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 $\mathcal{F}$:

$$\mathcal{F}_L : (\mathcal{F}M)N \rightarrow \mathcal{F}(\lambda k.M(\lambda m.k(mN))))$$
$$\mathcal{F}_R : V(\mathcal{F}M) \rightarrow \mathcal{F}(\lambda K.M(\lambda m.k(Vm))))$$

   "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:

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

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

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: $\lambda p\ \ f_{open}.(\mathbf{begin}(\#(p\ \ f_{open}))(\text{close}\ f_{open}))$ 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:

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

   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"

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

   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.

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.