Delimited Control

Stephen Chang

4/13/2010

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},
}
```

Summary

Haynes and Friedman show how to extend continuations and call/cc by first capturing the continuation provided by call/cc in a closure. The authors dub this closure a "continuation object (cob)" and it is passed to call/cc's argument. Capturing the continuation in a cob allows greater flexibility because additional operations can be performed before the continuation is invoked. Haynes and Friedman present several useful programs that utilize both cobs and call/cc variations implemented using cobs.

Cobs are used to implement fluid environments (dynamic binding). Fluid environments are used to implement dynamic domains, which are finally used to implement `dynamic-wind`. The `dynamic-wind` operator is used when a programmer wants to guarantee that certain operations are executed when a specified block of code is entered or exited. One application of `dynamic-wind` is to ensure that file streams are always properly closed when they are no longer needed.

Analysis

The primary conclusion of the paper is that continuations provided by call/cc must be properly harnessed and constrained in order to be useful. In the paper, cobs are used to accomplish this. However, the programs presented in the paper are rather inelegant and inefficient. The authors allude to this when they say that the paper "emphasized semantics, while ignoring some security and efficiency issues." The more general problem seems to be that call/cc is "too powerful" because most applications do not need the entire remainder of the program. A better control operator should give the programmer finer-grained access to the program's context.

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},
}
```

Summary

In this paper, Felleisen extends his λ_v–C calculus (λ_v–C is λ_v plus $\mathcal{F}$, a call/cc-like operator) with a delimiter # (also called a "prompt"). The original λ_v–C requires a special "computation" rule to eliminate $\mathcal{F}$ applications. The special rule is only applied when an $\mathcal{F}$ application is in the empty context, but this destroys the possibility of reasoning locally about terms using the calculus. With the prompt, expressions that are equal in the calculus behave equivalently in all contexts. Felleisen then derives an abstract machine from the new calculus, and also presents several programming applications of prompts, including a much simpler implementation of the `dynamic-wind` operator of Haynes and Friedman. Having explicit prompts in the language gives the programmer finer-grained control because on one hand, control operators can only capture context up to the nearest prompt delimiter, and on the other hand, control actions can only erase context up to the nearest prompt delimiter.

Analysis

It seems that the initial motivation for adding explicit prompts to the calculus was to fix operational equivalence, and it wasn't realized until afterwards that prompts are a useful programming construct in their own right. The primary benefit of the $\mathcal{F}$ and prompt operators is that control can be constrained, but another benefit (compared to an operator like call/cc) is that captured continuations can be composed. Perhaps this is implied (because an invocation of a

continuation in the abstract machine is just a stack append) but this does not seem to be explicitly stated in the paper. This paper is also an example that supports the study of formal language models. The λ_v–C revealed an alternative ($\mathcal{F}$) to an existing construct (call/cc), as well as uncovered a new control construct (prompts).

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},
}
```

Summary

A problem with Felleisen's $\mathcal{F}$ and `prompt` operators is that multiple uses of the operators in a program may interfere with each other. This interference affects not only the correct behavior of a program, but it is also a security issue because user-defined control operators may conflict with built-in constructs, thus crossing a programming language's abstraction boundary. After discussing $\mathcal{F}$ and `prompt` (called `control` and `run` this paper) and their relation to `call/cc`, as well as presenting several programming applications of `control` and `run`, Sitaram and Felleisen suggest that the interference problem can be addressed by installing a hierarchy of control operators. Specifically, each `control` and `run` is assigned a "level" and a `run` delimits all `control` applications at or above its level. Therefore, operators with a higher level have less access to the program context. In this way, a language can protect its abstractions while still providing the programmer with general `control` and `run` operators. It simply gives the programmer-level control operators a higher level. The authors add the qualification that their proposed scheme may not be ideal in every situation, but that any hierarchy of control operators would be superior to a flat world of control in enhancing the robustness and security of a language.

Abstracting Control

```
@inproceedings{Danvy1990AbstractingControl,
  author    = {Olivier Danvy and Andrzej Filinski},
  title     = {Abstracting Control},
  booktitle = {LISP and Functional Programming},
```

```
  year      = {1990},
  pages     = {151-160},
}
```

Summary

Danvy and Filinski use the denotational approach to describing the semantics of a call/cc-like operator. They also show that a hierarchy of delimited contexts can be achieved by repeatedly iterating the standard CPS transformation on a programming language (to get what they call "extended CPS"). The authors introduce the `shift` and `reset` operators, which provide the programmer access to the program context. Their two operators are comparable to the `control` and `prompt` operators of Felleisen and Sitaram. The `prompt` and `reset` delimiters have identical behavior. The one difference between `shift` and `control` is that the continuation captured by `shift` contains an implicit `reset`, whereas with the continuation captured `control` has no such delimter.

The authors present a denotational semantics of a language that includes `shift` and `reset`, as well as several applications of those control operators. One such application is implementing the CPS transformation metacircularly using the `shift` and `reset` operators themselves. This implementation also improves the efficiency of the CPS conversion because it folds in several optimizations that previously had to be performed after the conversion in a separate pass.

References