Ntinuation Copassing Style

Cameron Wong, Dez Reed

***Abstract*—We applied the categorical technique of "do that thing, but backwards" to continuation passing style to produce a novel programming idiom utilizing the `COME FROM` control flow operator.**

I. Introduction

Continuations are an important abstraction in the analysis and description of functional programs. By manipulating them, a savvy progrmamer can express complex control flow precisely and unambiguously. Unfortunately, with great power comes great responsibility, and code written in continuation-passing style is often obtuse and unreadable to those without such arcane knowledge. It is even said that CPS is merely an obfuscation technique designed to terrorize novice functional programmers and has no practical benefit.

Of course, obscure intermediate forms and programmer-facing implementations thereof need not be restricted to *functional* programs. A reasonable analogue may be the vaunted Static Single Assignment form, in which the programmer may need to use mystical φ-functions. As it turns out, however, SSA is actually equivalent to CPS and is therefore cheating. No, to achieve true parity, a different approach is needed.

Towards this end, we take a page from category theory, applying the time-worn approach of "flip everything backwards and see what happens". Monads embed impurity to a pure language, so too can comonads embed (enforced) purity into an impure language, and thus "ntinuation copassing style"* is born.

A continuation is a typesafe abstraction of the "return address". In other words, invoking a continuation is a type-safe delimited `GOTO`. A ntinuation, then, is the opposite of `GOTO`. For wisdom as to what, precisely, this entails, we turn to the programming language INTERCAL. While INTERCAL *does* have a form of `GOTO`, it also implements the `COME FROM` operator, which is precisely the opposite of `GOTO`!

What is copassing? The type of a CPS function in a traditional functional language typically involves `('a -> 'b) -> 'b`, which has arrows. Entering into the true categorical spirit and flipping the arrows, then, suggests that "copassing" a ntinuation must be typed at `('a <- 'b) <- 'b`. Unfortunately, this doesn't mean anything, but it is fun to write and look at.

*Categorically-inclined readers may protest that we have not properly complemented the "style". The authors believe that code written using `COME FROM` is already pretty much the opposite of "stylish", and thus is already complementary.

II. Ntinuations

In continuation passing style, the core observation is that, by passing the continuation as an argument, it allows the callee to manipulate it to construct new continuations and call them as necessary. In this way, the state of the overall program after any given subroutine invocation is neatly encapsulated into the continuation.

To flip this around, then, a ntinuation would be to increase expressiveness by removing control of not only the state of the program on subroutine return, but also the condition on which the return occurs.

A NCS-transformed function may contain no local variables, but instead relies entirely on its parameters and global, mutable state. The ntinuation, then, is subject to an exit condition imposed on the global state of the program. This process is known as "copassing" or "catching" the ntinuation. †.

This construction mimics the vaunted `COME FROM` control flow operator from the INTERCAL programming language in that a ntinuation represents a "trapdoor" that wrests control flow from the encapsulated routine on a given condition being met. In the true imperative spirit, ntinuations are given meaning entirely by global state and the manipulation thereof.

III. Example

Here is the factorial function, written in Ntinuation Copassing Style psuedo-code extended with ntinuations and subroutines:

```
ROUTINE fact:
  PLEASE NOTE that the input is passed in the
  1: SET y TO y * x
  SET x TO x-1
  PLEASE COME FROM (1) WITH y WHEN x EQUALS (
```

†An initial draft of this work was titled "termination catching style", which perhaps more neatly encapsulates this idea, but after much debate, we settled on "ntinuation" because it would be funnier watching people attempt to pronounce it.

In true INTERCAL fashion, the line beginning with `PLEASE NOTE` is a comment, as it begins with the text "PLEASE NOT"[‡].

This is the standard iterative formulation of the factorial function that multiplies the global result variable y by a loop counter x until x reaches 0. The key difference, however, is in the wrapped recursive call to `fact` that declares a new ntinuation which aborts computation and returns to that point.

Note that, should multiple exit conditions apply at once, it is nondeterministic which trapdoor opens, as in the original formulation in INTERCAL. In this case, all trapdoors lead to the same place with the same condition, so they will take control one after the other in some nondeterministic order, but do nothing before yielding control to another trapdoor. Finally, after all trapdoors are opened, the routine halts with the value of y containing the result of the factorial.

IV. Conclusion

We have presented a novel control flow operator in the style of continuation passing style by attempting to "reverse the arrows". In fact, the idea of conditional trapdoors (encapsulating `COME FROM`s) mimic that of pre-empting interrupts that take control from a process. In this way, just as CPS allows for precise manipulation of control flow by manipulating a subroutines' next steps, NCS does the reverse by moving all control flow into global condition triggers, allowing programmers to focus on the individual steps of a given operation. We expect this tool to become key in an imperative obfuscator's toolbox.

In future work, we hope to explore whether, as CPS forms a monad, NCS forms a true comonad (in fact, we believe that NCS *also* forms a monad, but we did not confirm this).

[‡]The authors originally intended to give this example in an extended INTERCAL, but gave up after four hours of whiteboarding a multiplication algorithm.