Categorical Structure of Continuation Passing Style

Hayo Thielecke

Doctor of Philosophy
University of Edinburgh
1997

Abstract

This thesis attempts to make precise the structure inherent in Continuation Passing Style (CPS).

We emphasize that CPS translates λ-calculus into a very basic calculus that does not have functions as primitive.

We give an abstract categorical presentation of continuation semantics by taking the continuation type constructor $\neg$ (or `cont` in Standard ML of New Jersey) as primitive. This constructor on types extends to a contravariant functor on terms which is adjoint to itself on the left; restricted to the subcategory of those programs that do not manipulate the current continuation, it is adjoint to itself on the right.

The motivating example of such a category is built from (equivalence classes of typing judgements for) continuation passing style (CPS) terms. The categorical approach suggests a notion of effect-free term as well as some operators for manipulating continuations. We use these for writing programs that illustrate our categorical approach and refute some conjectures about control effects.

A call-by-value λ-calculus with the control operator `callcc` can be interpreted. Arrow types are broken down into continuation types for argument/result-continuations pairs, reflecting the fact that CPS compiles functions into a special case of continuations. Variant translation are possible, among them "lazy" call-by-name, which can be derived by way of argument thunking, and a genuinely call-by-name transform. Specialising the semantics to the CPS term model allows a rational reconstruction of various CPS transforms.

Acknowledgements

I would like to thank my supervisors, Stuart Anderson and John Power.

Thanks for discussions and comments to Olivier Danvy, Matthias Felleisen, Andrzej Filinski, Masahito "Hassei" Hasegawa, John Hatcliff, Peter O'Hearn, Alan Paxton, Andy Pitts, Jon Riecke, David N. Turner and Phil Wadler.

Thanks to Glynn Winskel for inviting me to Aarhus for a week.

Diagrams were typeset with XY-pic.

I am grateful to my parents for moral and financial support throughout. To them this thesis is dedicated.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Hayo Thielecke*)

Table of Contents

List of Programs

Chapter 1

Introduction

The aim of this thesis is to make explicit the structure underlying continuation passing style, reifying it, so to speak, by making it less of a style and more of a structure.

There are (as yet) few programming languages that "have" continuations in the sense of possessing a language construct for giving unrestricted access to continuations. In a wider sense, however, most programming languages "have" continuations in some sense or another. In contemporary Computer Science, continuations may appear in various settings [1] and under different guises, among them at least the following:

- as a style of semantic definition in denotational semantics, giving meaning to generalised jumps;
- as a programming technique in mostly, or even purely, functional languages
- as a programming construct in (mostly/impurely) functional languages
- as a compiling technique

Many textbooks on denotational semantics, such as [Ten91] and [Sch86], contain some material on continuations in the context of imperative languages.

As a first-class continuation primitive is part of the official definition of Scheme [Re91], textbooks on Scheme, such as [FF96] typically give some examples of its use, the most advanced being perhaps [SF89].

The functional programming textbook [Hen87] gives a thorough introduction to the use of continuations in program transformation and code generation (interestingly, using a purely functional language without control operators — which

[1] We have listed here only what can be considered mainstream in that it appeared in several textbooks and is tought in undergraduate or at least MSc courses.

can be seen as evidence of the usefulness of continuations as a technique even without language support.)

Continuations as an implementation technique are used in [FWH92] for a toy interpreter and in [App92] for the New Jersey ML compiler.

1.1 An introduction to continuations in programming languages

This section is intended to provide some background: the reader familiar with continuations can safely skip it and jump to section 1.2 below.

The `goto` familiar from typical imperative (or "heritage") languages like C corresponds to a *command* continuation [SW74]. The much more powerful "jump with arguments", which we will be concerned with, corresponds to *expression* continuations. Here a value is passed, or "thrown" along with the transfer of control, much like the arguments in a function call. These were written with the special forms `valof` and `resultis` in [SW74]; this construct survives in typical imperative languages only in the case when the block is a function body: in this case the result of the function is `thrown` by the `return` statement.

Incidentally, the reason, in our view, that `goto` may justly be considered "harmful" [Dij68] is that it is so *weak*. In particular, it cannot pass arguments and can be compared to `GOSUB` (without arguments) as a cruder and less structured counterpart of a genuine procedure call.

See figures 1.1, 1.2 and 1.3 on page 8 for examples of jumps with and without arguments.

We are particularly interested in two aspects of continuations: their use in giving semantics to control operators, and their use in compiling functions into more primitive jumps with arguments.

We illustrate the use of the control operator `callcc` by discussing a simple example, using both Scheme and (the New Jersey version of) ML in the hope that the reader may be familiar with one of these.

For Scheme, first-class continuations are part of the language definition [Re91]. In ML they are not, but the New Jersey implementation (see the manual [NJ93]) adds first-class continuations to ML by means of the following signature:

```
type 'a cont
val callcc : ('1a cont -> '1a) -> '1a
val throw : 'a cont -> 'a -> 'b
```

```
char* f()
{
  return "Threw past the loop.\n";
  while(1);
}

main()
{
  goto skip;
  while(1);
  skip: printf("Jumped past the loop.\n");
  printf("%s", f());
}
```

Figure 1.1: Two jumps in C

```
fun loop x = loop x;

callcc(fn skip =>
       loop(throw skip ()));
output(std_out,"Jumped past the loop.\n");

output(std_out,
       callcc(fn skip =>
              loop(throw skip "Threw past the loop.\n")));
```

Figure 1.2: Two jumps in ML

```
(define (loop x) (loop x))

(begin
  (call/cc(lambda (skip)
            (loop (skip (list)))))
  (write "Jumped past the loop."))

(write
  (call/cc(lambda (skip)
            (loop (skip "Threw past the loop.")))))
```

Figure 1.3: Two jumps in Scheme

```
fun remberuptolast a lat =
    callcc(fn skip =>
       let fun R [] = []
           |   R (b::l) =
               if b = a then throw skip (R l) else b::(R l)
       in
           R lat end);
```

Figure 1.4: `remberuptolast` in ML

1.1.1 An example program

As an example of the use of expression continuations in programming, we consider the function `rember-upto-last` form the recent programming textbook *The Seasoned Schemer* [FF96]:

> *The function* `rember-upto-last` *takes an atom a and a lat [list of atoms] and removes all the atoms from the lat up to and including the last occurrence of a. If there are no occurrences of a,* `rember-upto-last` *returns the list.*

First we transliterate [2] the function `rember-up-to-last` from the original Scheme to ML (see figure 1.4 on page 9).

`remberuptolast a lat` removes everything up to the last occurrence of `a` from the list `lat`. For instance:

```
- remberuptolast 42 [];
val it = [] : int list
- remberuptolast 42 [1,2,3,4,5,6,7,8,9];
val it = [1,2,3,4,5,6,7,8,9] : int list
- remberuptolast 42 [1,2,3,4,42,5,6,7,8,9];
val it = [5,6,7,8,9] : int list
- remberuptolast 42 [1,2,3,4,42,5,6,7,42,8,9];
val it = [8,9] : int list
- remberuptolast 42 [1,2,3,4,42,5,6,7,42,8,9,42];
val it = [] : int list
```

[2]We pass between the different lexical conventions for identifiers in ML abd Scheme (e.g.`callcc` and `call/cc`) without emphasising it. Similarly, in a more idealised setting we write λ where one would have `fn` in ML and `lambda` in Scheme.

```
fun remberuptolast a lat =
    callcc(fn skip =>
           let fun R [] = ()
                 | R (b::l) =
                   (R l;
                    if b = a then throw skip l else ())
           in
               (R lat; lat) end);
```

Figure 1.5: `remberuptolast` without consing in ML

```
(define (rember-upto-last a lat) ; Look Ma no cons
  (call/cc
   (lambda (skip)
     (letrec
         ((R
           (lambda (l)
             (if (null? l)
                 (list)
                 (begin
                   (R (cdr l))
                   (if (eq? a (car l))
                       (skip (cdr l))
                       (list)))))))
     (begin (R lat) lat)))))
```

Figure 1.6: `remberuptolast` without consing in Scheme

The local helper function R in remberuptolast recurs over the list lat; every time the element a is encountered the remainder of the list is made the overall result by being passed to the result continuation skip. Note that this is essentially iteration and jumping, which becomes even clearer if we rewrite remberuptolast so that it does not copy the list (figures 1.5 and 1.6 for remberuptolast without consing). In this version, R does all its work by recurring and jumping, its return value being irrelevant. In that sense, it is highly non-functional, but rather "jumperative"[3]. In fact, we can push the analogy with imperative programming even further. The solution of traversing the list and throwing every time an a in found is similar to the imperative solution of dragging a pointer across the list, that is, updating a variable every time an a is found (figures 1.7 and 1.8). In some sense, there is a duality here: a throw *preempts later* throws, while an assignment *undoes earlier* assignments. Similarly, providing lat as a default (for the case when no jump occurs) result at the very end of the argument of callcc is analogous to initialising p to lat at the very beginning.

In this example, the jump with arguments provided by continuation invocation could also be written using ML [Pau91], [MTH90] exceptions (which may be more familiar to some readers): see figure 1.9. callcc(fn skip => ...) is roughly analogous to declaring a local exception and and handling it by retaining the value it passed; while while throwing a value to a continuation is analogous to raising an exception with that value.

Exceptions, and their semantic differences compared with first-class continuations, are beyond the scope of this thesis. If pushed too far, the analogy with exceptions may actually be misleading.

Hence we consider another way of explicating the role of continuations in the example: that is, by making them an explicit argument to a function. This foreshadows the formal (continuation) semantics of callcc.

Figures 1.10 and 1.11 show how remberuptolast can be written by explicitly passing a continuation parameter during the recursion of R. This yields a purely functional program, as all occurrences of control operators have been expanded out or compiled away, as it were.

Note how jumping (if an a is found in the list) in the programs with callcc in figures 1.5 and 1.6 amounts to ignoring the continuation parameter k in the programs with an explicit continuation parameter in figures 1.10 and 1.11. [4]

[3] to use a term coined by M-x dissociated-press in emacs

[4] Experts would perhaps point out that the versions of rember-upto-last with explicit continuations in figures 1.10 and 1.11 are not strictly the continuations counterparts of the ones with callcc and that they correspond more closely to aborting whenever an a is found and

```
fun remberuptolast a lat =
    let val p = ref lat in
        let fun R [] = ()
              | R (x::l) =
                    (if x = a then p := l else ();
                     R l)
        in
            R lat;
            !p
        end
    end;
```

Figure 1.7: `remberuptolast` with dragging a pointer across the list (In ML)

```
(define (rember-upto-last a lat)
  (letrec
      ((p lat)
       (R
        (lambda (l)
          (if (not (null? l))
              (begin
                (if (eq? a (car l))
                    (set! p (cdr l)))
                (R (cdr l)))))))
    (begin
      (R lat)
      p)))
```

Figure 1.8: `remberuptolast` with dragging a pointer across the list (In Scheme)

```
fun remberuptolastexn a lat =
    let exception skipexn of int list in
        let fun R [] = ()
              | R (b::l) =
                (R l;
                 if b = a then raise skipexn l else ())
        in
            (R lat; lat)
        end
    handle skipexn x => x
    end;
```

Figure 1.9: `remberuptolast` (without consing) with ML exceptions

```
fun remberuptolast a lat =
    let fun R [] k = k ()
          | R (b::l) k =
            R l
            (fn () =>
             if b = a then l else k ())
    in
        R lat (fn () =>  lat) end;
```

Figure 1.10: `remberuptolast` with explicit passing of a continuation parameter

```
(define (rember-upto-last a lat) ; Look Ma, no cons
  (letrec
      ((R
        (lambda (l k) ; R has a continuation parameter k
          (if (null? l)
              (k)
              (R (cdr l)
                 (lambda ()
                   (if (eq? a (car l))
                       (cdr l)
                       (k)))))))))
    (R lat (lambda () lat))))
```

Figure 1.11: `remberuptolast` with explicit passing of a continuation parameter

While this formulation is very succinct, it is hard to understand in intuitive programming terms, in that what is intuitively a jump is expressed by modifying the current continuation before passing it to the recursive call. Here we have introduced continuations only in that small portion of the program that makes use of them. In general, we would have to introduce them everywhere, leading to a virtually unreadable program full of anonymous λ terms representing continuations.

1.1.2 Upward continuations

The example of `rember-upto-last` in section 1.1.1 is perhaps not totally felicitous in that it only uses "downward" continuations (in the sense of [FWH92]: a continuation can be passed "down"into a function call as an argument, but not "up" from it as a result). Downward continuations do not really reveal the full power of first-class continuations, as the latter comprise also the "upward" case. The last two chapters of the textbook [SF89] are devoted to the use of continuations; unfortunately heavy use is made of local state. While this combination of first-class continuations with local state gives rise to a very useful programming idiom (coroutines), it does not illustrate the power of continuations on their own, without state. It is not totally clear whether, in the absence of local state to encapsulate the current continuation [HFW86], [SF89], or at least a global queue of suspended threads [BCL+96], one can obtain coroutines from continuations. See also [Shi96].

Nevertheless, the following, somewhat Mickey Mouse, example of two coroutines from [SF89] can be written without using state. Instead we use a function `phi` to manipulate the current continuation, about which we will say more in section 1.2 below.

```
(define (phi f)
  (lambda (h)
    (call/cc (lambda (k) (h (f k))))))

(define (ping a)
  (phi (lambda (x) (write a) x)))

(((ping 'ping) (ping 'pong)) ((ping 'ping) (ping 'pong)))
```

The two functions printing `ping` and `pong`, respectively, call each other inces-

delimiting this within the definition of `rember-upto-last`.

santly, producing a string

`pingpongpingpongpingpong...`

In fact, an even more distilled construction of looping from a double self-application with first-class continuations in the untyped setting of Scheme is witnessed by the fact that

```
((call/cc call/cc) (call/cc call/cc))
```

loops.

The failure to consider upward continuations appears to be the cause of a misapprehension that one sometimes encounters, holding that "there are no closed terms of continuation type". It is easy to find examples refuting this, e.g. the following:

```
- callcc(fn k => throw (callcc(fn h => throw k h)) 42);
val it = cont : int cont cont
```

Below, we introduce a set of functions (1.12 on 19 and figure 1.13 on page 20 for ML and Scheme) with which we can write terms of continuation type more succinctly, for instance

```
- thunk 42;
val it = cont : int cont cont
```

— as well as the following:

```
- funtocont (fn x : int => x);
val it = cont : (int * int cont) cont
```

1.1.3 Continuation Passing Style

In the last version of `remberuptolast` (figures 1.10 and 1.11) the function `R` takes an explicit continuation argument instead of seizing the current continuation by means of a `callcc`. More generally, a program with `callcc` can be translated into one without, but such that everything takes an explicit continuation argument. (This continuation argument is an ordinary function, not an element of some special continuation type.) For a highly idealised programming language, namely simply-typed λ-calculus augmented by the constant `callcc`, this translation is the CPS transform (adapted here from [DHM91], an extension of [Plo75]):

$$
\begin{array}{rcl}
\overline{x} & = & \lambda k.kx \\
\overline{\lambda x.M} & = & \lambda k.k(\lambda xh.\overline{M}h) \\
\overline{MN} & = & \lambda k.\overline{M}(\lambda f.\overline{N}(\lambda x.fxk)) \\
\overline{\mathtt{callcc}\ M} & = & \lambda k.\overline{M}(\lambda f.fkk) \\
\overline{\mathtt{throw}\, M\, N} & = & \lambda k.\overline{M}\,(\lambda k.\overline{N}k)
\end{array}
$$

CPS transforms such as this may seem quite confusing if considered formalistically as translations of λ-calculi, in that there is no obvious sense in which they are homomorphic or otherwise structure-preserving; certainly λ is not translated to λ.

This becomes somewhat clearer if we think of a CPS transform as an idealised compilation. For instance, the translation of a function $\lambda x.M$ needs an explicit return address h for the function body M along with its argument x; that explains $\lambda xh.\overline{M}h$. As we have a language with higher-order functions, the whole function needs to be computed in the first place. Now function definitions evaluate to themselves, or more accurately, the compiled $\lambda xh.\overline{M}h$ is immediately passed to the overall return address k, i.e. $k(\lambda xh.\overline{M}h)$.

In a sense, no function in CPS ever returns; each will ultimately call some continuation. So in order to make CPS work, only some mechanism for passing arguments (without returning) is required — such as input prefixes in a process calculus. What is crucial, though, is that the recipient address of such an input can itself be passed as an argument. But that is the main feature of the π-calculus (where this is often called "mobility").

We can thus transliterate the CPS transform above, yielding a transform with the π-calculus as the target language, related to, though not quite identical with, Milner's translations [PS93]. (The main reason that Milner's translation differs from our is that it was originally designed for the monadic, not the polyadic, variant of the π-calculus.) In addition, this transform also has clauses for `callcc` and `throw`:

$$
\begin{array}{rcl}
(\!|x|\!)(k) & = & \overline{k}\langle x\rangle \\
(\!|\lambda x.M|\!)(k) & = & (\nu l)(\overline{k}\langle l\rangle \mid {!}\, l(xh).(\!|M|\!)(h)) \\
(\!|\mathtt{throw}\ M\ N|\!)(k) & = & (\nu m)((\!|M|\!)(m) \mid {!}\, m(n).(\!|N|\!)(n)) \\
(\!|\mathtt{callcc}\ M|\!)(k) & = & (\nu m)((\!|M|\!)(m) \mid {!}\, m(l).\overline{l}\langle kk\rangle) \\
(\!|MN|\!)(k) & = & (\nu m)((\!|M|\!)(m) \mid {!}\, m(l).(\nu n)((\!|N|\!)(n) \mid {!}\, n(a).\overline{l}\langle ak\rangle))
\end{array}
$$

1.1.4 CPS as name-passing

We briefly comment on CPS as an idealised compilation, using Milner's π-calculus as the target language.

Exponentials and λ-abstraction are often taken as foundational for the semantics of programming languages. Naively, though, a "function" call consists of two jumps with arguments: first the caller jumps to the callee, passing it the actual parameters (if any) and the return address; the callee jumps to the return address, passing the result (if any).

Less naively, one could argue that a concept of reference, address or pointer (here in particular: pointer to code) is more fundamental for actual computation than the notion of function; the π-calculus is perhaps the most successful embodiment of this view. But it is chiefly the mobility aspect of the π-calculus that matters here, concurrency being somewhat orthogonal.

Both continuation-passing style [Plo75, DHM91] and the π-calculus decompose or "compile" function abstraction into such jumps with arguments. Recall the clause for λ:

$$(\!|\lambda x.M|\!)(k) = (\nu l)(\overline{k}\langle l\rangle \mid !\, l(xh).(\!|M|\!)\langle h\rangle)$$

Here the caller would have to jump to the address l with actual parameters for x and h, while the callee, for, say $M = a$, would jump to h with argument a. For simplicity, let us consider a function without arguments, i.e. a delayed term or thunk—the control flow becomes clearer if the jumping is not interspersed with arguments.

$$(\!|\mathtt{delay} M|\!)(k) = (\nu l)(\overline{k}\langle l\rangle \mid !\, l(h).(\!|M|\!)(h)h)$$

Such delayed terms can be forced to evaluate themselves by sending them a request for data, that is, an address where these are to be sent.

$$(\!|\mathtt{force}\,(\mathtt{delay}\,M)|\!)(k) = (\nu m)\left((\!|\mathtt{delay}\,M|\!)\langle m\rangle \mid m(l).\bar{l}\langle k\rangle\right)$$

So in particular, for $M = a$,

$$\begin{array}{rcl}
(\!|\mathtt{force}\,(\mathtt{delay}\,a)|\!)(k) & = & (\nu m)\left((\!|\mathtt{delay}\,a|\!)(m) \mid !\, m(l).\bar{l}\langle k\rangle\right) \\
& = & (\nu m)\left((\nu l)(\overline{m}\langle l\rangle \mid !\, l(h).\overline{h}\langle a\rangle) \mid !\, m(l).\bar{l}\langle k\rangle\right) \\
& = & (\nu l)\left(!\, l(h).\overline{h}\langle a\rangle \mid \bar{l}\langle k\rangle\right) \\
& = & \overline{k}\langle a\rangle \\
& = & (\!|a|\!)(k)
\end{array}$$

Looking at the jumps $\overline{m}\langle l\rangle$, $\bar{l}\langle k\rangle$, $\overline{k}\langle a\rangle$, we could observe that control first flows outward, as the delayed expression evaluates to the thunk located at l, then inwards, as the `force` sends the thunk a request to evaluate itself, and then outward again, if the forced thunk sends a. From the point of view of the `delayed` expression, though, one could equally say that its surrounding evaluation context is delayed—the thunk $(!\, m(l).\bar{l}\langle k\rangle)$ located at m— and needs to be `force`d by the $\overline{m}\langle l\rangle$. The complementary forcings $\overline{m}\langle l\rangle$ and $\bar{l}\langle k\rangle$ cancel each other out. Somehow the computation seems to be turned around, or even to be turned inside out (twice, even). This appears to have inspired the coinage "pivots"[PS96].

In our view, this is not an epiphenomenon, but something characteristic of control flow in a mostly functional setting. In fact, we will base our account on a categorical notion of turning inside out, that is duality in the sense of adjointness of a functor to its own dual.

1.2 Introducing the continuation functor

We give a first exposition of the crucial concepts from a programming perspective.

All important examples, displayed in figures, are bilingual, in both ML and Scheme. In the main text there is usually a certain bias towards ML, largely because ML produces type information along with results.

The ML implementation used in the experiments was Standard ML of New Jersey, Version 0.93. No ML implementation with *both* the new value polymorphism and first-class continuations was available at the time of writing; that is why we have weak type variables (`'1a`, `'2b`, ...) in programs with continuations (see [HDM93] and [NJ93]). For Scheme, Bigloo (v1.6) and Gambit were used.

A by-product of a categorical semantics is a set of so-called categorical combinators. For λ-calculus, its categorical semantics in Cartesian closed categories yields, for instance, the evaluation map and morphism pairing. These can be seen as constituting a combinatory logic, with the added benefit of being more semantically inspired than the Schönfinkel combinators S and K (see [FH88] for a discussion). It is in fact easier to define the negation functor if we know that what we are aiming for is an adjunction. Both the isomorphism of adjunction ϕ and the unit $\mathsf{force}$are easy to define, and in terms of these, we have $\neg f = \phi(f \circ \mathsf{force})$,as in figures 1.12 and 1.13. Defined in one step, the negation functor is somewhat harder to read, not least because of the nested `callcc`:

```
fun negate f =
    fn h => callcc(fn k =>
                   throw h (f (callcc(fn p =>
                                      throw k p))));
```

```
fun force h = callcc(throw h);
    force : '1a cont cont -> '1a;

fun phi f h = callcc((throw h) o f);
    phi : ('2a cont -> 'b) -> ('b cont -> '2a);

fun negate f = phi(f o force);
    negate : ('1a -> 'b) -> ('b cont -> '1a cont);

fun thunk a = callcc(fn k => throw (force k) a);
    thunk : '1a -> '1a cont cont;

fun conttofun c a =
    callcc(fn k => throw c (a,k));
    conttofun : ('a * '2b cont) cont -> ('a -> '2b);

fun funtocont f =
    callcc((fn (a,k) => throw k (f a)) o force);
    funtocont : ('1a -> '1b) -> ('1a * '1b cont) cont;

fun delay f x = ((negate(negate f)) o thunk) x;
    delay :  ('2a -> '2b) -> ('2a -> '2b cont cont);
```

Figure 1.12: Categorical combinators for continuations in NJ-SML

```
negate : ('2a -> 'b) -> ('b cont -> '2a cont);
```

The typing, such as it is, of continuations in Scheme consist of the single axiom

$$\texttt{(call-with-current-continuation procedure?)} \Rightarrow \texttt{\#t}$$

stating that what `call-with-current-continuation` passes to its argument is a procedure.

It is essential here that continuations do not have to be unary, that is they can take more than one argument — this makes it possible to identify functions with a special case of continuations. In ML, this can be accommodated easily, as a multi-argument function or continuation is one that takes a tuple of arguments. Moreover, this is symmetric in that multiple return values amount to a single return value that is a tuple.

Writing the same programs in Scheme is slightly awkward, because standard "R4RS" (as specified by the Revised[4] Report on the Algorithmic Language

```scheme
(define compose
  (letrec ((compose-list
            (lambda (l)
              (lambda (x)
                (if (null? l)
                    x
                    ((car l) ((compose-list (cdr l)) x)))))))
    (lambda l
      (compose-list l))))

; force (in our sense) = call/cc when applied to a continuation

(define (phi f)
  (lambda (h)
    (call/cc (compose h f))))

(define (negate f)
  (phi (compose f call/cc)))

(define (cont-to-fun c)
  (lambda (a)
    (call/cc (lambda (k)
               (c (list a k))))))

(define (fun-to-cont f)
  (call/cc
   (compose
    (lambda (ak)
      ((cadr ak) (f (car ak))))
    call/cc)))

(define (thunk a)
  (call/cc (lambda (k)
             ((call/cc k) a))))
```

Figure 1.13: Categorical combinators for continuations in Scheme

Scheme [Re91]), does not have multiple return values. In order to return multiple values, one needs to return a single list.

Hence, unlike in ML, we cannot use even use the rudimentary pattern matching available for `lambda` expressions in Scheme

$$(\texttt{lambda}\ (x_1 \ldots x_n)\ M)$$

Instead, we write only single-argument procedures whose argument is a list. The individual arguments are extracted from this list using projections `car`, `cadr`, caddr, ...

In more modern implementations of Scheme incorporating multiple return values as proposed in [Ree92], one could rewrite most of the Scheme programs presented here more elegantly.

In Scheme, there is some possibility of confusion between the `force` as we define it and the built-in procedure `force` in Scheme. In fact, both have nearly the same functionality of forcing a thunk. The difference between them is that thunk forced by the Scheme `force` is call-by-need, in that a second forcing will not evaluate it again, whereas our `force` in terms of continuations only (without any updating) conforms to the original call-by-name meaning of thunks [Ing61].

1.2.1 The self-adjointness of higher-order jumping

We give a detailed, but not formalised, argument in terms of what happens during evaluation. (One could formalise this, either using a CPS transform, or the operational semantics from [HDM93].)

We would like to show that the structure that we wish to analyse is due to the intended meaning, independent of any particular formalisation.

Let `callcc` and `throw` be abbreviated as `C` and `T`, respectively.

From a programming perspective, the self-adjointness is closely related to a style of using continuations that Sabry calls Continuation Grabbing Style [Sab96].

Its characteristic idiom is the following pattern of use of `callcc`:

$$(\lambda x.N)(\mathtt{C}(\lambda k.M))$$

where both N and M jump out of their context by ultimately throwing. We say that the `callcc` binds k to the continuation $\lambda x.N$.

`force` allows the argument of a continuation to turn the tables on its continuation.

$$\mathtt{T}h(\mathtt{force}\,k) \quad = \quad \mathtt{T}\,k\,h$$

$\mathsf{force} = \lambda k.\mathtt{C}(\lambda p.\mathtt{T}\,k\,p)$

$$
\begin{aligned}
& \mathtt{T}\,h\,(\mathtt{force}\,k) \\
= \;& \mathtt{T}\,h\,(\mathtt{C}(\lambda p.\mathtt{T}\,k\,p))
\end{aligned}
$$

$\mathtt{T}\,h$ makes h the continuation of its argument. $\mathtt{T}\,k\,h$

$$
\begin{aligned}
& \neg(\lambda x.x) \\
= \;& \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h((\lambda x.x)\mathtt{force}\,k)) \\
= \;& \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h((\lambda x.x)(\mathtt{C}(\lambda p.\mathtt{T}\,k\,p)))) \\
= \;& \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h(\mathtt{C}(\lambda p.\mathtt{T}\,k\,p))) \\
= \;& \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,k\,h)
\end{aligned}
$$

$$
\begin{aligned}
& \neg f \circ \neg g \\
= \;& (\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\,h_1\,f\,(\mathtt{C}(\lambda p_1.\mathtt{T}\,k_1\,p_1))))(\lambda h_2.\mathtt{C}(\lambda k_2.\mathtt{T}\,h_2\,g\,(\mathtt{C}(\lambda p_2.\mathtt{T}\,k_2\,p_2)))) \\
= \;& \lambda h_2.(\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\,h_1\,f\,(\mathtt{C}(\lambda p_1.\mathtt{T}\,k_1\,p_1))))(\mathtt{C}(\lambda k_2.\mathtt{T}\,h_2\,g\,(\mathtt{C}(\lambda p_2.\mathtt{T}\,k_2\,p_2))))
\end{aligned}
$$

$(\lambda h_1 \ldots)$ is a λ-expression, so its argument is evaluated next. The `callcc` binds the k_2 to $\lambda h_1 \ldots$ and $\mathtt{T}\,g$ makes the argument position of g the continuation of the following term; this seizes its current continuation and binds it to p_2, which is then `thrown` to k_2. Because k_2 was bound to $\lambda h_1 \ldots$ it follows that h_1 becomes p_2. The $\mathtt{C}(\lambda k_1 \ldots)$ seizes the overall continuation and binds k_1 to it. $\mathtt{T}\,h_1$ makes h_1 the continuation for f. Hence if f returns a result, this will be fed to h_2 and thus to g. The argument position of f is seized by the $\mathtt{C}(\lambda p_1 \ldots)$, and bound to p_1 and then `thrown` to k_1.

What is important here is that the `throwing` in the middle amounts just to a function composition of f and g, so the whole term is equivalent to

$$\lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h\,(g \circ f)(\mathtt{C}(\lambda p.\mathtt{T}\,k\,p)))$$

which is $\neg(g \circ f)$.

$$
\begin{tikzcd}
\neg\neg A \arrow[r, "\mathsf{force}"] \arrow[d, "\neg\neg f"'] & A \arrow[d, "f"] \\
\neg\neg B \arrow[r, "\mathsf{force}"] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
\neg A \arrow[r, "\neg\mathsf{force}"] \arrow[dr, "\mathsf{id}"'] & \neg\neg\neg A \arrow[d, "\mathsf{force}"] \\
& \neg A
\end{tikzcd}
$$

For example

```
- 1+callcc(fn k => 10 + throw ((force o (negate force)) k) 2);
val it = 3 : int
```

$$
\begin{aligned}
\texttt{force} &= \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h\,k) \\
\neg f &= \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h\,(f\,(\texttt{force}\,k))) \\
&= \lambda h.\mathtt{C}(\lambda k.\mathtt{T}\,h\,(f\,(\mathtt{C}(\lambda p.\mathtt{T}\,k\,p))))
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{force} \circ \neg\texttt{force} \\
= \;& (\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\,h_1\,k_1)) \\
& \circ(\lambda h_2.\mathtt{C}(\lambda k_2.\mathtt{T}\,h_2\,(\texttt{force}\,(\mathtt{C}(\lambda p.\mathtt{T}\,k_2\,p))))) \\
= \;& (\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\,h_1\,k_1)) \\
& \circ(\lambda h_2.\mathtt{C}(\lambda k_2.\mathtt{T}\,h_2\,((\lambda h_3.\mathtt{C}(\lambda k_3.\mathtt{T}\,h_3\,k_3))\,(\mathtt{C}(\lambda p.\mathtt{T}\,k_2\,p))))) \\
= \;& \lambda h_2.(\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\,h_1\,k_1))(\mathtt{C}(\lambda k_2.\mathtt{T}\,h_2\,((\lambda h_3.\mathtt{C}(\lambda k_3.\mathtt{T}\,h_3\,k_3))\,(\mathtt{C}(\lambda p.\mathtt{T}\,k_2\,p)))))
\end{aligned}
$$

If we try to formulate the triangular identity in prose, we would arrive at the following narrative about jumping:

Let us assume that $\texttt{force} \circ \neg\texttt{force}$ gets evaluated. h_2 becomes the current argument; let us call the overall continuation of the whole expression k. We would like to show that all that happens is that, in some circuitous manner, the current argument h_2 is passed to the current continuation k.

First of all, the term in the operator position is evaluated; as it is a λ-expression $(\lambda h_1 \ldots)$, its argument is evaluated next. This is has `C` in the operator expression: evaluating $\mathtt{C}(\lambda k_2 \ldots)$ it binds k_1 to $\lambda h_1 \ldots$. Then the `throw` to h_2 is executed, making h_2 the continuation of the subsequent term. This is an application, with a λ-expression $(\lambda.k_3 \ldots)$ in the operator position, so its argument is evaluated next. Again, executing the $\mathtt{C}(\lambda p \ldots)$ binds p to $\lambda h_3 \ldots$, before `throw`ing it to k_2. Now k_2 was bound to λh_1, so h_1 becomes p. The body of the λ-expression then executes the `callcc` in $\mathtt{C}(\lambda k_1 \ldots)$, which binds k_1 to the current continuation, which at this point is the overall continuation k. This is then `thrown` to h_1. This having been bound to p, which in turn points to $\lambda.h_3 \ldots$, h_3 becomes k. The body of the λ-expression following λh_3 is then evaluated; this is the fourth and last `callcc`. This $\mathtt{C}(\lambda k_3 \ldots)$ binds k_3 to the current continuation, which, due to the surrounding $\mathtt{T}\,h_2$, is just h_2. Finally, this is `thrown` to k_3, which is to say to the overall continuation k.

Now we turn to the naturality.

$$\begin{array}{rl} & f \circ \mathtt{force} \\ = & \lambda h. f(\mathtt{C}(\lambda k. \mathtt{T}\, h\, k)) \end{array}$$

The thunk h is forced and the result supplied to f as its argument.

$$\begin{array}{rl} & \mathtt{force} \circ \neg\neg f \\ = & (\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\, h_1\, k_1)) \\ & \circ\ (\lambda h_2.\mathtt{C}(\lambda k_2.\mathtt{T}\, h_2\, ((\lambda h_3.\mathtt{C}(\lambda k_3.\mathtt{T}\, h_3\, (f\ (\mathtt{force}\, k_3)))\ (\mathtt{force}\, k_2))) \\ = & (\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\, h_1\, k_1)) \\ & \circ\ (\lambda h_2.\mathtt{C}(\lambda k_2.\mathtt{T}\, h_2\, ((\lambda h_3.\mathtt{C}(\lambda k_3.\mathtt{T}\, h_3\, (f\ (\mathtt{C}(\lambda p_3.\mathtt{T}\, k_3\, p_3)))))\ (\mathtt{C}(\lambda p_2.\mathtt{T}\, k_2\, p_2))))) \\ = & \lambda h_2.(\lambda h_1.\mathtt{C}(\lambda k_1.\mathtt{T}\, h_1\, k_1)) \\ & \ (\mathtt{C}(\lambda k_2.\mathtt{T}\, h_2\, ((\lambda h_3.\mathtt{C}(\lambda k_3.\mathtt{T}\, h_3\, (f\ (\mathtt{C}(\lambda p_3.\mathtt{T}\, k_3\, p_3)))))\ (\mathtt{C}(\lambda p_2.\mathtt{T}\, k_2\, p_2))))) \end{array}$$

h_2 is the overall argument. The operator position is $(\lambda h_2. \ldots)$; hence the argument is evaluated. This is a `callcc`, which binds k_2 to $\lambda h_1. \ldots$. The $\mathtt{T}\, h_2$ is executed, making h_2 the continuation of the following term. The operator is $(\lambda k_3. \ldots)$, so the argument is evaluated. This is $\mathtt{C}(\lambda p_2. \ldots)$, which binds p_2 to $\lambda k_3. \ldots$. The $\mathtt{T}\, k_2$ is executed; because k_2 was bound to $\lambda h_1. \ldots$, it follows that h_1 becomes p_2. The body of $\lambda h_1. \ldots$ is evaluated; $\mathtt{C}(\lambda k_1. \ldots)$ binds k_1 to the overall continuation k. This is then passed to h_1. Because h_1 was bound to p and p to $\lambda h_3. \ldots$, this means that h_3 becomes k. The $\mathtt{C}(\lambda k_3. \ldots)$ binds k_3 to the current continuation at this point. Because of the surrounding $\mathtt{T}\, h_2$ this is h_2.

Hence the whole term is equivalent to

$$\lambda h_2. f(\lambda p_3.\mathtt{T}\, h_2\, p_3)$$

Not only do first-class continuations give rise to an adjunction; this is also a particularly simple kind of adjunction. Whereas one would normally have two functors, two naturality squares (one each for unit and counit) and two triangular identities comprising an adjunction (as in a Cartesian closed category, say), we have one of each. (This is fair enough somehow, in that a continuation is half a function.)

1.2.2 Alternative control operators

We explain that our categorical combinators give a complete set of control operators, and hence an alternative to `callcc` and `throw`.

In the previous section, we focussed on the functor and the unit. But an adjunction can equally well be expressed by the isomorphisms of adjunction; we now explain how this can be seen as a new control operator.

`phi` together with a coercion function from functions to continuations is a complete set of control operators, like `callcc` together with its coercion function `throw` (coercing $\neg\tau$ to $\tau \to \alpha$).

`callcc` does two conceptually quite separate things with the current continuation: first it copies it, then it makes one of the copies available to its argument as an ordinary function argument. The other copy is given as the current continuation to the argument of `callcc`.

One could separate these; in particular Felleisen's control operator $\mathcal{C}$ does not copy the continuation. The continuation is given as an argument to the argument of $\mathcal{C}$, but the current continuation is *not* supplied to it.

Like $\mathcal{C}$, `phi` considered as a control operator does not copy the continuation, without the need to consider terms that can do without the current continuation.

For comparison, we list the CPS semantics of

- `callcc` with ML-style typing;
- a variant `call/cc` closer to that of Scheme, in that the continuation is wrapped into a procedure;
- the $\mathcal{C}$-operator, which is like `call/cc`, but does not copy the continuation it seizes;
- ϕ, or `phi` in ASCII, which does not copy the continuation either, but requires a second argument to supply the continuation for its first.

$$\begin{array}{rcl}
\overline{\texttt{callcc}\,M} & = & \lambda k.\overline{M}(\lambda f.fkk) \\
\overline{\texttt{call/cc}\,M} & = & \lambda k.\overline{M}(\lambda f.f(\lambda x k'.kx)k) \\
\overline{\mathcal{C}M} & = & \lambda k.\overline{M}(\lambda f.f(\lambda x k'.kx)(\lambda x.x)) \\
\overline{\phi\,M\,N} & = & \lambda k.\overline{M}(\lambda f.\overline{N}(\lambda n.fkn))
\end{array}$$

For comparison: $\texttt{callcc}(\lambda k.M)$ binds the current continuation, which nonetheless is also the continuation for M, to k; $\mathcal{C}(\lambda k.M)$ binds the current continuation to k, the price for which is that M does not get a current continuation; $\phi(\lambda k.M)\,h$ binds the current continuation to k and supplies h as the continuation for M.

None of these is any more generic than the others, as they are all interdefinable [5], but ϕ is perhaps special in that it emphasises a certain symmetry: both `callcc` and `throw` are special instances of it.

In the typing of continuations in ML, `callcc` needs its companion `throw`, which is just a coercion from continuations to functions.

If one is willing to identify continuations with certain procedures those that ignore their result continuation), as in Scheme, then `callcc` on its own is enough.

If we are willing to make another identification, reducing functions to continuations, rather than embedding continuations into functions, then `phi` on its own, without coercions, is enough. In continuation semantics, a function call consists of passing both an argument and a return continuation to a function. A function, then, is just something that expects these two: in other words, a continuation for an argument/result continuation pair.

Hence, if $\sigma \to \tau$ were an abbreviation for $\neg(\sigma * \neg\tau)$, `phi` by itself would be sufficient. We can recover `callcc` and `force` from `phi` as follows.

```
fun throw2 a = phi(fn h => a);
    throw2 : 'a -> 'a cont -> '2b;

fun callcc2 f = (phi(fn k => (k,k))) (funtocont f);
    callcc2 : ('1a cont -> '1a) -> '1a;
```

The unit `force` and the negation functor can be defined similarly in terms of `phi`.

```
val force2 = phi(fn x => x);
    force2 : '1a cont cont -> '1a;

fun negate2 f = phi(f o force2);
    negate2 : ('1a -> 'b) -> 'b cont -> '1a cont;
```

Axiomatising, and calculating with, `phi`, `force` and `negate` would then be guided by the standard equational laws for adjunctions.

1.3 Related Work

Variants of the continuation functor, though not *qua* functor, have made appearances in the literature, e.g. in [Hof94]. The following from [Shi96] also appears to be related.

[5] We gloss over the issue of the aborting implicit in $\mathcal{C}$.

```
(compose-cont k f)  ≡  (lambda (v) (f v k))
```

By the standards of Computer Science, and particularly among advanced programming language concepts, continuations are of great antiquity: the term "continuation" was coined in [SW74]; Continuation Passing Style appears implicitly in [Fis72] (final version in [Fis93]) and explicitly in [Ste78]. (For a history see [Rey93].)

They are also (explicitly or implictly) an almost ubiquitous concept. Thus the potential background to the present thesis is vast. In addition to the literature on continuations proper e.g. [Plo75, FFKD86, SF90, HDM93], work on the π-calculus ([Mil91] and [PS93]; explicitly in [Bou97]) and Scheme [Ste77, Ste78, Ste76] also has some relevance.

A " deep" connection between continuations and classical logic in sometimes claimed, e.g. [Gri90], [RS94]. This seems orthogonal to our approach. Or we could argue that first-class continuations have plenty of interesting structure in their own right, so that there is no pressing need to establish connections to logic in order to elucidate them.

There is, however, a much smaller area of work that is of direct relevance here. The first attempt at a categorical continuation semantics was Filinski's pioneering [Fil89]. With the benefit of hindsight, [Fil92] is in its emphasis on linearity an aberration. Filinski later chose to regard continuations not as primitive but as a special instance of monads [Fil96]. Similar in its use of the monadic metalanguage to provide a systematic presentation and classification is Danvy and Hatcliff's [DH94]. Much can be done in that setting, but decomposing the monad into two instances of the continuation functor affords a more fine-grained analysis — including, crucially, the control operators and an abstract account of thunking [HD95].

Finally, one of the most important influences was the typing of first-class continuations in Standard ML of New Jersey, with the continuation type as primitive [HDM93]. (See also the conference version [DHM91], where a CPS semantics is given.) This type discipline is a natural starting point for a (categorical) semantics: looking for universal properties of the continuation type constructor, one is led to self-adjointness.

Incidentally, it was primarily for reasons of polymorphic typing that the continuation type was made primitive in Standard ML of New Jersey [HDM93] (whereas the more minimalist, and untyped, Scheme simply conflates continuations and procedures). We should like to regard this as a fortunate preadaption (in the Darwinian sense) on the part of ML.

To summarise, what is perhaps amazing about Continuation Passing Style is how far one can get with three little equations

$$\begin{array}{rcl}\overline{x} & = & \lambda k.kx \\ \overline{\lambda x.M} & = & \lambda k.k(\lambda xk.\overline{M}k) \\ \overline{MN} & = & \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mnk))\end{array}$$

These from [Plo75], together with two more if we include `callcc` and `throw` [DHM91], encapsulate much of the backdrop to this thesis. To distill things further, we could say that the essence of the transformation is really in the clause for $\overline{\lambda x.M}$, that is, what happens to a function. Much of the effort herein is devoted to trying to understand what

$$\overline{\lambda x.M} = \lambda k.k(\lambda xk.\overline{M}k)$$

really means, without taking the λ's on the right too literally, but rather adopting the view point of "λ the ultimate `goto`" [Ste77].

1.4 Outline

The aspect of CPS that is particularly emphasized in this thesis is that it breaks down function types into continuations.

This view of function calls as "jump with arguments" [Ste78, SS76] is not low-level and implementation specific, but should be taken seriously in semantics.

We also develop a calculus in support of this view. The categorical account should be seen as complementary, not as an alternative, to it. The bureaucracy inherent in names and their scope is particularly virulent in a name-passing calculus, and although the CPS calculus is in some sense like an internal language, even conceptually primitive operations can have quite complicated representations (for instance `thunk` and `pair`). This makes the more high-level, variable free, perspective of a categorical description a valuable addition.

Focussing on the category of computations also facilitates experimentation, in that we can write programs in real world languages, without some monadic interpreter. as a bag on the side of Haskell, say.

In such experiments, or validations of concepts, the categorical semantics suggests building blocks (for instance $\circ$ for functions, `map` for lists [Bac78]), which could be regarded as "categorical combinators", like `eval` for the λ-calculus.

In our case, the use of these categorical combinators lets us avoid spaghetti code, like nested occurrences `callcc`.

Just as we try to be faithful to those features of CPS that are in evidence, such as breaking down of functions types, we avoid introducing anything that is not naturally part of it. A case in point are coproducts, in particular, the empty coproduct $\mathbf{0}$ and the identification of continuations with functions $A \to \mathbf{0}$.

Parts of this thesis have appeared in [Thi96a] [Thi97]; some of it is joint work [PT97], comprising chapter 8 here.

Chapter 2 The target language of the CPS transforms is presented as a calculus in its own right, which we call the CPS calculus. This calculus is very simple and quite low-level: only variables may be passed as arguments, moreover an application is more like a jump with arguments than a λ-calculus application in that it forgets its calling context. Compared to λ-calculus, the CPS calculus could be said to be somewhere in between the λ-calculus itself and explicit substitutions. Some variants are also considered, mostly for theoretical reasons.

Chapter 3 is a review and discussion of various CPS transforms that have appeared in the literature. Call-by-value is the basic case, various other calling mechanisms being derivable by argument thunking.

Chapter 4 The categorical account of the structure underlying continuation semantics is developed. Its fundamental structure is what we call self-adjointness, i.e. a functor adjoint to its own dual in the two possible senses, i.e. on the left and on the right. Environments are modelled by means of premonoidal structure. This comes equipped with a notion of *central* morphism.

Chapter 5 A term model is constructed as an instance of the categorical framework in chapter 4. This is a CPS analogue of the construction of a Cartesian closed category from simply-typed λ-calculus. In the setting of the term model, the syntactic form of CPS terms can be related to the semantic properties of the morphisms they represent.

Chapter 6 is an excursion, inasmuch as it illustrates some issues concerning (semantic) notions of effect-freeness by means of concrete examples and counter-examples. Specifically, we demonstrate that a term being *cancellable* (which has also been called *total*) is not sufficient for it to be free of effects, whereas it being *central* is. At the same time, it is a first attempt at showing how the categorical structure of continuations can help to write programs, as we build on the functions defined in section 1.2. Thus it complements chapter 5, using the structure from chapter 4 at the level of the *source* language of CPS transforms.

Chapter 7 The categorical counterpart of the CPS transforms is given by categorical semantics. Parts of this chapter parallel chapter 3, giving a rational reconstruction of CPS transforms. Among the categorical structures introduced in Chapter 4, the self-adjointness on the left is shown to underlie both the semantics of control operators and the `thunk`/`force`-mechanism for variant calling strategies.

Chapter 8 A different categorical perspective on the self-adjointness is provided by studying it in the framework of indexed categories; this shows the fundamental structure for continuations to be independent of the way environments are modelled. Initially, in early drafts, there were two separate formulations of the categorical continuation semantics presented in this thesis; these were then shown to be essentially equivalent in joint work with John Power.

Chapter 9 We present some (preliminary) material on graphical representations capturing some aspects of CPS. The formal link is again established by self-adjointness — which can be visualised in this setting as turning upside down. Some issues concerning the relation of CPS to duality are raised.

Chapter 10 concludes and points towards directions for further work, among them some loose ends from the previous chapters as well as some more ambitious proposals giving continuations a fundamental rôle.

The reader interested chiefly in the programming perspective may find it useful to concentrate on section 1.2 and Chapter 6. The latter can be understood independently of most of the preceeding chapters. It is quite long, because many programs are included, but also because the counterexamples presented there, while initially intended to show only that a certain subcategory does not admit a (canonical) product, proved quite fruitful in refuting many naive assumption about continuations.

Some knowledge of category theory would probably be helpful, but only very little is really required. The various equivalent characterisations of adjunctions (found in any category theory textbook, e.g. [Mac71]) would perhaps be the most useful thing to keep in mind.

Chapter 2

The CPS calculus

We consider the target language of CPS transforms as a calculus in its own right (similar to the intermediate language of the compiler in [App92]), which we call the CPS calculus.

The CPS calculus was first used as a common idiom for the λ- and the π-calculus in [Thi96b]. It was then turned into a calculus in its own right in the course of initial discussions with Phil Wadler and David N. Turner.

Notational preliminaries

We let lowercase letters $x, y, n, m, k, l, \ldots$ range over variables (names) and uppercase letters $M, N, \ldots$ range over terms (in various calculi). $\vec{x}, \vec{y}, \ldots$ range over sequences $x_1 \ldots x_i$ of names. Commas in sequences are often omitted. When used as indices, lowercase letters range over natural numbers, e.g. $x_1 \ldots x_n$.

We write $M[x \mapsto N]$ for the capture-avoiding substitution of N for x in M. Similarly, if $\vec{x} = x_1 \ldots x_j$ and $\vec{y} = y_1 \ldots y_j$, we write $M[\vec{x} \mapsto \vec{y}]$ for the simultaneous substitution of y_i for x_i $(i = 1, \ldots, j)$ in M.

We use the traditional semantics brackets $[\![-]\!]$ for (categorical) semantics and the slightly different $(\!|-|\!)$ parentheses for transformations that can be seen as somewhat intermediate between a proper semantics and a mere translation (a matter of degree, not principle).

2.1 CPS calculus

The raw terms of the CPS calculus are given by the following BNF:

$$M ::= \quad x\langle\vec{x}\rangle \mid M\{x\langle\vec{x}\rangle{=}M\}$$

We call a term of the form $k\langle\vec{x}\rangle$ a *jump* and a term of the form $M\{n\langle\vec{x}\rangle{=}N\}$ a *binding*. As a first hint at the intended meaning, $k\langle\vec{x}\rangle$ is a jump to the continuation k with actual parameters $\vec{x}$, while $M\{n\langle\vec{x}\rangle{=}N\}$ binds the continuation with body N and formal parameters $\vec{x}$ to n in M.

2.1.1 Remark While succinct, the presentation of the syntax is strictly speaking an abuse of notation, common for λ-calculi, in that the same symbol is used for a syntactic category and the typical metavariable ranging over it. A more technically orthodox BNF could be given as follows.

$$\begin{array}{rcl} Term & ::= & Var \texttt{"}\langle\texttt{"}\ Var^*\ \texttt{"}\rangle\texttt{"} \\ & | & Term\ \texttt{"}\{\texttt{"}\ Var\ \texttt{"}\langle\texttt{"}\ Var^*\ \texttt{"}\rangle{=}\texttt{"}\ Term\ \texttt{"}\}\texttt{"} \end{array}$$

Every CPS term can be written as a jump followed by a sequence of bindings, that is, a term of the following form

$$k\langle\vec{x}\rangle\{p_1\langle\vec{y}_1\rangle{=}M_1\}\dots\{p_n\langle\vec{y}_n\rangle{=}M_n\}$$

Hence the BNF could be written in a somewhat cluttered form, like this:

$$M ::= x\langle\vec{x}\rangle\{x\langle\vec{x}\rangle{=}M\}\dots\{x\langle\vec{x}\rangle{=}M\}$$

The set of free variables $\mathsf{FV}(M)$ of a CPS term M is defined as follows.

$$\begin{array}{rcl} \mathsf{FV}(x\langle y_1\dots y_k\rangle) & = & \{x, y_1, \dots, y_k\} \\ \mathsf{FV}(M\{n\langle y_1\dots y_k\rangle{=}N\}) & = & (\mathsf{FV}(M)\setminus\{n\})\cup(\mathsf{FV}(N)\setminus\{y_1,\dots,y_k\}) \end{array}$$

In $M\{n\langle\vec{x}\rangle{=}N\}$ the scope of n extends to the left, while that of the x_i extends to the right. Therefore we have left and right α-conversions.

$$\begin{array}{rcll} M\{n\langle\vec{x}\rangle{=}N\} & = & M\{n\langle\vec{y}\rangle{=}N[\vec{x}\mapsto\vec{y}]\} & (\alpha\mathrm{R}) \\ M\{n\langle\vec{x}\rangle{=}N\} & = & M[n\mapsto n']\{n'\langle\vec{x}\rangle{=}N\} & (\alpha\mathrm{L}) \end{array}$$

We usually gloss over the α-conversion by identifying terms up to renaming of bound variables.

The axioms of CPS calculus are as follows.

$$\begin{array}{rcll} L\{m\langle\vec{x}\rangle{=}M\}\{n\langle\vec{y}\rangle{=}N\} & = & L\{n\langle\vec{y}\rangle{=}N\}\{m\langle\vec{x}\rangle{=}M\{n\langle\vec{y}\rangle{=}N\}\} & (\textsc{Distr}) \\ & & m\neq n \quad m,\vec{x}\notin\mathsf{FV}(N) & \\ k\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} & = & k\langle\vec{y}\rangle, \quad n\notin\mathsf{FV}(k\langle\vec{y}\rangle) & (\mathrm{GC}) \\ n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} & = & N[\vec{z}\mapsto\vec{y}] & (\textsc{Jmp}) \\ M\{n\langle\vec{x}\rangle{=}n'\langle\vec{x}\rangle\} & = & M[n\mapsto n'] & (\textsc{Eta}) \end{array}$$

The ($\textsc{Jmp}$) law is in some sense what drives the computation. By contrast, (GC) and ($\textsc{Distr}$) can be seen as "structural" laws like those of the π-calculus.

Most of these laws appear in Appel's [App92]. See also [Ste78].

We will be concerned primarily with simply-typed CPS terms. The only type constructor is the negation type $\neg(\_)$. The BNF for (simple) types for the CPS calculus is as follows.

$$\tau ::= \neg(\tau_1 \dots \tau_n) \mid b$$

where b ranges over base types.

Terms are then typed according to these two rules:

$$\frac{}{\Gamma, k : \neg\vec{\tau}, \vec{y} : \vec{\tau} \vdash k\langle\vec{y}\rangle} \qquad \frac{\Gamma, n : \neg\vec{\tau} \vdash M \quad \Gamma, \vec{y} : \vec{\tau} \vdash N}{\Gamma \vdash M\{n\langle\vec{y}\rangle{=}N\}}$$

Typing judgements in CPS are one-sided. Whereas in λ-calculus a judgement $\Gamma \vdash_\lambda M : \tau$ (where we have decorated the Urteilsstrich $\vdash$ with a λ to emphasise that this is a λ-calculus judgement) states that, in the type environment Γ, the term M has the type τ, a CPS typing judgement $\Gamma \vdash M$ states that, in the type environment Γ the CPS term is consistent. Similar type systems exist for process calculi, e.g. [Tur95]. For example, $x : \tau \vdash_\lambda x : \tau$ states that under the assumption that x has type τ, x has type τ. By contrast $x : \tau, k : \neg\tau \vdash k\langle x\rangle$, states that, under the assumption that x has type τ and k has type $\neg\tau$, passing x to τ "does not go wrong".

Alternatively, one could compare a CPS term to a *command* in languages like Idealized Algol, in that it is run for effect, not value.[1] In that sense, a CPS judgement $x_1, \dots, x_n \vdash M$ is analogous to M being a command, as in

$$x_1 : \mathbf{var}[\tau_1], \dots, x_n : \mathbf{var}[\tau_n] \vdash M : \mathbf{comm}$$

We could call this "consistency" in that it implies that all internal communications channels, so to speak, are used in a consistent manner. Logically it would appear closer to *inconsistency*, inasmuch as $\Gamma \vdash M$ could be read as "M witnesses that Γ entails a contradiction", such as in the example above where Γ contained both the assumptions that τ and not τ. The typing rule for the binding construct could then be read as stating that if both "Γ and $\vec{\tau}$" and "Γ and not $\vec{\tau}$" entail a contradiction, then the contradiction must be due to Γ alone.

In addition to these rules, we assume permutation, contraction and weakening of typing environments unless explicitly stated otherwise.

$$\frac{x_1 : \tau_1, \dots, x_n : \tau_n \vdash M}{x_{\pi(1)} : \tau_{\pi(1)}, \dots, x_{\pi(n)} : \tau_{\pi(n)} \vdash M} \qquad \pi \text{ is a permutation of } \{1, \dots, n\}$$

$$\frac{\Gamma \vdash M}{\Gamma, x : \tau \vdash M} \qquad \frac{\Gamma, x : \vec{\tau}, y : \vec{\tau} \vdash M}{\Gamma, x : \vec{\tau} \vdash M[y \mapsto x]}$$

[1]This was pointed out to me by Peter O'Hearn.

One could visualize a CPS term as representing the state of a (stackless, heap-allocating) abstract machine.

$$\underbrace{q}_{\text{IP}}\underbrace{\langle p_1 \dots p_l\rangle}_{\text{actuals}}\{\underbrace{n_1}_{\text{address}}\underbrace{\langle x_1 \dots x_i\rangle}_{\text{formals}}=\underbrace{N_1}_{\text{code}}\}\dots\{n_m\langle\vec{x}_m\rangle=N_m\}$$

2.2 Recursive CPS calculus

All that is needed in order to make the calculus recursive is to change the visibility of names, making the address of a binding visible within its body, so that in $M\{n\langle\vec{x}\rangle\Leftarrow N\}$, N may refer to itself under n.

We use a slightly different notation for the binding construct, "$\Leftarrow$" instead of "$=$", to indicate the possibility of recursion.

$$M ::= \quad x\langle\vec{x}\rangle \mid M\{x\langle\vec{x}\rangle\Leftarrow M\}$$

Again, the BNF could be rendered in a more orthodox fashion:

$$\begin{array}{rcl} Term & ::= & Var\ \texttt{"}\langle\texttt{"}\ Var^*\ \texttt{"}\rangle\texttt{"} \\ & | & Term\ \texttt{"}\{\texttt{"}\ Var\ \texttt{"}\langle\texttt{"}\ Var^*\ \texttt{"}\rangle\Leftarrow\texttt{"}\ Term\ \texttt{"}\}\texttt{"} \end{array}$$

For the recursive CPS calculus, we modify the typing as follows.

$$\frac{\Gamma, n : \neg\vec{\tau} \vdash M \quad \Gamma, \vec{x} : \vec{\tau}, n : \neg\vec{\tau} \vdash N}{\Gamma \vdash M\{n\langle\vec{x}\rangle\Leftarrow N\}} \quad \text{Cyclic Closure}$$

As we have broadened the scope of n, we need to modify the left α-conversion correspondingly.

$$M\{n\langle\vec{y}\rangle\Leftarrow N\} \quad \equiv \quad M[n \mapsto n']\{n'\langle\vec{y}\rangle\Leftarrow N[n \mapsto n']\}$$

The definition of free variables needs to be modified similarly.

$$\begin{array}{rcl} \mathsf{FV}(x\langle y_1 \dots y_k\rangle) & = & \{x, y_1, \dots, y_k\} \\ \mathsf{FV}(M\{n\langle y_1 \dots y_k\rangle\Leftarrow N\}) & = & (\mathsf{FV}(M) \setminus \{n\}) \cup (\mathsf{FV}(N) \setminus \{n, y_1, \dots, y_k\}) \end{array}$$

The set of bound variables is defined as follows.

$$\begin{array}{rcl} \mathsf{BV}(x\langle y_1 \dots y_k\rangle) & = & \emptyset \\ \mathsf{BV}(M\{n\langle y_1 \dots y_k\rangle\Leftarrow N\}) & = & \mathsf{BV}(M) \cup \mathsf{BV}(N) \cup \{n, y_1, \dots, y_k\}) \end{array}$$

The axioms of the recursive CPS-calculus are as follows.

$$\begin{array}{rcll} L\{m\langle\vec{x}\rangle\Leftarrow M\}\{n\langle\vec{y}\rangle\Leftarrow N\} & = & L\{n\langle\vec{y}\rangle\Leftarrow N\}\{m\langle\vec{x}\rangle\Leftarrow M\{n\langle\vec{y}\rangle\Leftarrow N\}\} & (\textsc{Distr}) \\ & & m \neq n \quad m, \vec{x} \notin \mathsf{FV}(N) & \\ k\langle\vec{y}\rangle\{n\langle\vec{z}\rangle\Leftarrow N\} & = & k\langle\vec{y}\rangle, \quad n \notin \mathsf{FV}(k\langle\vec{y}\rangle) & (\text{GC}) \\ n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle\Leftarrow N\} & = & N[\vec{z} \mapsto \vec{y}]\{n\langle\vec{z}\rangle\Leftarrow N\} & (\textsc{RecJmp}) \\ M\{n\langle\vec{x}\rangle\Leftarrow n'\langle\vec{x}\rangle\} & = & M[n \mapsto n'] & (\textsc{RecEta}) \\ & & n \neq n' & \end{array}$$

Notice that the modification of the closure typing law makes the unrestricted ETA law unsound. Consider

$$n\langle a\rangle\{x\langle x\rangle{\Leftarrow}n\langle x\rangle\} = n\langle a\rangle[n \mapsto n] = n\langle a\rangle.$$

But we know that $n\langle a\rangle\{x\langle x\rangle{\Leftarrow}n\langle x\rangle\}$ loops, which should not be identified with the terminating $n\langle a\rangle$. Hence the side condition on variables precluding the looping.

2.3 Operational semantics for CPS

We consider jumping as the only behaviour of CPS terms. The jumping axiom is accompanied by the distributive law, which can be seen as a structural congruence (in the sense of Milner) or "heating" (in the chemical metaphors of Boudol), in that its purpose is to bring together the component parts of a (jumping) redex.

2.3.1 Definition We define oriented versions of the axioms DISTR and JUMP as follows (avoiding name capture):

$$\begin{array}{rcll} L\{m\langle\vec{x}\rangle{=}M\}\{n\langle\vec{y}\rangle{=}N\} & \rightharpoonup & L\{n\langle\vec{y}\rangle{=}N\}\{m\langle\vec{x}\rangle{=}M\{n\langle\vec{y}\rangle{=}N\}\} & ,n \neq m \\ n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} & \rightarrow & N[\vec{z} \mapsto \vec{y}] & \end{array}$$

2.3.1 Observational congruence

While derivability from the CPS axiom is the least equality we would wish to impose on CPS terms, a notion of observational equivalence is arguably the greatest such notion we could consider. (Here least and greatest are to be understood in the sense of set-theoretic inclusion of relations, i.e. the least equality equates the fewest terms.)

We define observational equivalence $\approx$ for CPS. We choose at our notion of observation the "external" jump that a term may perform after it has performed some internal jumps. For instance, in a jump of the form $k\langle\vec{x}\rangle$, we can observe (the occurrence of a a jump to) k. More generally, a free variable in the leftmost position can be observed.

2.3.2 Definition Let $M \downarrow k$ iff $M \rightharpoonup^* \rightarrow^* k\langle\vec{x}\rangle\{p_1\langle\vec{y}_1\rangle{=}M_1\}\ldots\{p_n\langle\vec{y}_n\rangle{=}M_n\}$ and $k \notin \{p_1, \ldots, p_n\}$.

Let $M \approx N$ iff for all contexts C and names k, $C[M] \downarrow k$ iff $C[N] \downarrow k$

The axioms of the equational theory of the CPS calculus become laws in the operational congruence.

2.3.3 Proposition

$$
\begin{array}{rcll}
L\{m\langle\vec{x}\rangle{=}M\}\{n\langle\vec{y}\rangle{=}N\} & \approx & L\{n\langle\vec{y}\rangle{=}N\}\{m\langle\vec{x}\rangle{=}M\{n\langle\vec{y}\rangle{=}N\}\} & (\textsc{Distr}) \\
 & & m \neq n \quad m, \vec{x} \notin \mathsf{FV}(N) & \\
k\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} & \approx & k\langle\vec{y}\rangle, \quad n \notin \mathsf{FV}(k\langle\vec{y}\rangle) & (\mathrm{GC}) \\
n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} & \approx & N[\vec{z} \mapsto \vec{y}] & (\textsc{Jmp}) \\
M\{n\langle\vec{x}\rangle{=}n'\langle\vec{x}\rangle\} & \approx & M[n \mapsto n'] & (\textsc{Eta})
\end{array}
$$

This follows from preliminary work by Massimo Merro on a restricted version of the π-calculus [Davide Sangiorgi and Massimo Merro, personal communication]; the details may appear elsewhere.

2.4 Linear CPS calculus

We consider a linear version of the calculus, as linearity will sometimes allow a less complicated account.

In the linear CPS calculus the Garbage Collection and the Distributivity axioms do not make sense. For a term $M\{n\langle y\rangle{=}N\}$, GC is applicable iff n does not occur in M, while Distr allows one to distribute the binding for n to multiple occurrences. Neither of these cases is well-typed in the linear calculus. So instead we have two separate axioms allowing us to "float" a binding into a term $L\{M\langle x\rangle{=}M\}$, depending upon whether n occurs in the left subterm (L) or the right one (M). The linear calculus is still a fragment of the general CPS calculus, as each application of the floating laws can be simulated by distributing and garbage collection.

2.4.1 Definition The axioms of the linear CPS calculus are as follows.

$$
\begin{array}{rcll}
n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} & = & N[\vec{z} \mapsto \vec{y}] & (\textsc{Jmp}) \\
M\{n\langle\vec{x}\rangle{=}n'\langle\vec{x}\rangle\} & = & M[n \mapsto n'] & (\textsc{Eta}) \\
L\{m\langle\vec{x}\rangle{=}M\}\{n\langle\vec{y}\rangle{=}N\} & = & L\{n\langle\vec{y}\rangle{=}N\}\{m\langle\vec{x}\rangle{=}M\} & (\textsc{Float-L}) \\
 & & \text{if } m \neq n, n \notin \mathsf{FV}(M) & \\
L\{m\langle\vec{x}\rangle{=}M\}\{n\langle\vec{y}\rangle{=}N\} & = & L\{m\langle\vec{x}\rangle{=}M\{n\langle\vec{y}\rangle{=}N\}\} & (\textsc{Float-R}) \\
 & & \text{if } m \neq n, n \notin \mathsf{FV}(L) &
\end{array}
$$

The main point of restrictions like linearity is that they allow translations from CPS to less powerful calculi: the linear CPS calculus can be translated into Milner's action structure for the π-calculus [Mil93]. Let $k\langle\tilde{x}\rangle^{\dagger} = (\tilde{y})[\overline{k}\langle\tilde{x}\rangle]\langle\rangle$. If $M^{\dagger} = (\tilde{y}n)S\langle\rangle$ and $N^{\dagger} = (\tilde{y})T\langle\rangle$, then let

$$(M\{n\langle\tilde{x}\rangle{=}N\})^{\dagger} = (\tilde{y})[\nu n]^{\frown}S^{\frown}[n(\tilde{x})]^{\frown}T\langle\rangle$$

2.4.1 Linear unary CPS calculus

But we are more interested in restricting the linear CPS calculus even further. The subset of CPS calculus in which only a single argument is allowed in jumps $k\langle x\rangle$ and bindings $M\{n\langle x\rangle{=}N\}$ is called unary, as opposed to polyadic. The BNF for the unary subset of the CPS calculus is this:

$$M ::= \quad x\langle x\rangle \mid M\{x\langle x\rangle{=}M\}$$

For the fragment of CPS calculus that is both linear and unary, we can give a simplified presentation of the typing rules:

$$\frac{}{xk \vdash k\langle x\rangle} \qquad \frac{}{xk \vdash x\langle k\rangle}$$

$$\frac{xn \vdash M \quad yk \vdash N}{xk \vdash M\{n\langle y\rangle{=}N\}} \qquad \frac{nk \vdash M \quad yx \vdash N}{xk \vdash M\{n\langle y\rangle{=}N\}}$$

2.4.2 Definition The axioms of the linear unary CPS calculus are as follows.

$$\begin{array}{rcll}
n\langle y\rangle\{n\langle z\rangle{=}N\} & = & N[z \mapsto y] & (\textsc{Jmp}) \\
M\{n\langle \vec{x}\rangle{=}n'\langle \vec{x}\rangle\} & = & M[n \mapsto n'] & (\textsc{Eta}) \\
L\{m\langle \vec{x}\rangle{=}M\}\{n\langle y\rangle{=}N\} & = & L\{n\langle y\rangle{=}N\}\{m\langle \vec{x}\rangle{=}M\} & (\textsc{Float-L}) \\
 & & \text{if } m \neq n, n \notin \mathsf{FV}(M) & \\
L\{m\langle \vec{x}\rangle{=}M\}\{n\langle y\rangle{=}N\} & = & L\{m\langle \vec{x}\rangle{=}M\{n\langle y\rangle{=}N\}\} & (\textsc{Float-R}) \\
 & & \text{if } m \neq n, n \notin \mathsf{FV}(L) &
\end{array}$$

2.4.3 Remark In a binding expression $M\{n\langle \vec{x}\rangle{=}N\}$, the bindings of N and $\vec{x}$ are conceptually quite distinct: one could try to reflect this in the calculus by letting them bind variables from different zones of the type environment.

$$\frac{\Gamma; \vec{x}, \Delta \vdash M}{\Gamma, \vec{x}; \Delta \vdash M} \qquad \frac{\Gamma, \vec{x}; \Delta \vdash M}{\Gamma; \vec{x}, \Delta \vdash M}$$

$$\frac{}{\Gamma; \vec{x}, k \vdash k\langle \vec{x}\rangle} \qquad \frac{\Gamma, n : \neg\tau; \Delta \vdash M \quad \Gamma; \vec{x}, \Delta \vdash N}{\Gamma; \Delta \vdash M\{n\langle \vec{x}\rangle{=}N\}}$$

2.5 Constants

Although they play no rôle in the sequel, we sketch how PCF-style constants for arithmetic and conditionals could be added to the CPS calculus. Constants, like everything in CPS, take a continuation parameter.

$$\frac{}{\Gamma, n : \mathtt{int}, f : \neg\neg\vec{\tau}, g : \neg\neg\vec{\tau}, k : \neg\vec{\tau} \vdash \mathtt{ifzero}\langle \ulcorner n{+}1 \urcorner f g k\rangle} \qquad \frac{}{\Gamma, k : \neg\mathtt{int} \vdash \mathtt{0}\langle k\rangle}$$

$$\frac{}{\Gamma, n : \mathtt{int}, k : \neg\mathtt{int} \vdash \mathtt{succ}\langle \ulcorner n \urcorner k\rangle} \qquad \frac{}{\Gamma, n : \mathtt{int}, k : \neg\mathtt{int} \vdash \mathtt{pred}\langle \ulcorner n \urcorner k\rangle}$$

$$
\begin{array}{lcl}
\mathtt{0}\langle k\rangle & = & k\langle\ulcorner 0\urcorner\rangle \\
\mathtt{succ}\langle\ulcorner n\urcorner k\rangle & = & k\langle\ulcorner n{+}1\urcorner\rangle \\
\mathtt{pred}\langle\ulcorner n\urcorner k\rangle & = & k\langle\ulcorner n{-}1\urcorner\rangle \\
\mathtt{ifzero}\langle\ulcorner 0\urcorner fgk\rangle & = & f\langle k\rangle \\
\mathtt{ifzero}\langle\ulcorner n{+}1\urcorner fgk\rangle & = & g\langle k\rangle \\
\mathtt{diverge}\langle k\rangle & = & p\langle\rangle\{p\langle\rangle{\Leftarrow}p\langle\rangle\}
\end{array}
$$

2.6 Translation from CPS calculus

If we think in terms of α-equivalence classes of terms, then the non-recursive CPS calculus is evidently a subset of the recursive one, as we can embed terms of the former in the latter

$$M\{n\langle\vec{x}\rangle{=}N\} \quad \mapsto M\{n\langle\vec{x}\rangle{\Leftarrow}N\}$$

provided we α-convert in case n is free in N to avoid its name-capturing in $M\{n\langle\vec{x}\rangle{\Leftarrow}N\}$.

The recursive CPS calculus is in turn a fragment of Appel's intermediate language, the main difference being that Appel's `FIX` constructor allows *mutual* recursion.

CPS-calculus	Appel's datatype `cexp`
$x\langle y_1 \dots y_j\rangle$	`APP(VAR` x`,[VAR` y_1`,`$\dots$`,VAR` y_j`])`
$M\{n\langle x_1 \dots x_j\rangle{\Leftarrow}N\}$	`FIX([(`n`,[`x_1`,`$\dots$`,`x_j`],`N`)],`M`)`

The binding of continuations in CPS can be implemented not only by "passing" (using the λ-calculus), but equally by "sending" (π-calculus) or even "grabbing" (using `callcc` to seize the current continuation [Sab96]).

First, the (recursive) CPS calculus can be translated into simply-typed λ-calculus with a fixpoint combinator.

$$
\begin{array}{rcl}
k\langle x_1 \dots x_n\rangle^\circ & = & kx_1 \dots x_n \\
(M\{n\langle\vec{x}\rangle{\Leftarrow}N\})^\circ & = & (\lambda n.M^\circ)(\mu n.\lambda\vec{x}.N^\circ)
\end{array}
$$

Here μ is a fixpoint-finder in the simply-typed λ-calculus satisfying $\mu x.M = M[x \mapsto \mu x.M]$ and subject to the following typing

$$\frac{\Gamma, f : \tau \vdash M : \tau}{\Gamma \vdash \mu f.M : \tau}$$

2.6.1 Proposition The translation $(\_)^\circ$ is sound.

Proof

$$
\begin{array}{rl}
 & (n\langle\vec{x}\rangle\{n\langle\vec{y}\rangle{\Leftarrow}N\})^\circ \\
= & (\lambda n.n\,\vec{x})(\mu n.\lambda\vec{y}.N^\circ) \\
= & (\lambda n.n\,\vec{x})((\lambda\vec{y}.N^\circ)[n\mapsto\mu n.\lambda\vec{y}.N^\circ]) \\
= & (\lambda\vec{y}.N^\circ[n\mapsto\mu n.\lambda\vec{y}.N^\circ])\,\vec{x} \\
= & N^\circ[n\mapsto\mu n.\lambda\vec{y}.N^\circ][\vec{y}\mapsto\vec{x}] \\
= & N^\circ[\vec{y}\mapsto\vec{x}][n\mapsto\mu n.\lambda\vec{y}.N^\circ] \\
= & (\lambda n.N^\circ[\vec{y}\mapsto\vec{x}])(\mu n.\lambda\vec{y}.N^\circ) \\
= & (\lambda n.N[\vec{y}\mapsto\vec{x}]^\circ)(\mu n.\lambda\vec{y}.N^\circ) \\
= & (N[\vec{y}\mapsto\vec{x}]\{n\langle\vec{y}\rangle{\Leftarrow}N\})^\circ
\end{array}
$$

Let $n\neq n'$.

$$
\begin{array}{rl}
= & (M\{n\langle\vec{x}\rangle{\Leftarrow}n'\langle\vec{x}\rangle\})^\circ \\
= & (\lambda n.M^\circ)(\mu n.\lambda\vec{x}.n'\,\vec{x}) \\
= & (\lambda n.M^\circ)((\lambda\vec{x}.n'\,\vec{x})[n\mapsto\mu n.\lambda\vec{x}.n'\,\vec{x}]) \\
= & (\lambda n.M^\circ)(\lambda\vec{x}.n'\,\vec{x}) \\
= & (\lambda n.M^\circ)\,n' \\
= & M^\circ[n\mapsto n'] \\
= & (M[n\mapsto n'])^\circ
\end{array}
$$

$$
\begin{array}{rl}
 & (M\{n\langle\vec{x}\rangle{\Leftarrow}N\}\{l\langle\vec{y}\rangle{\Leftarrow}L\})^\circ \\
= & (\lambda l.(\lambda n.M^\circ)(\mu n.\lambda\vec{x}.N^\circ))(\mu l.\lambda\vec{y}.L^\circ) \\
= & (\lambda n.M^\circ)(\mu n.\lambda\vec{x}.N^\circ))[l\mapsto\mu l.\lambda\vec{y}.L^\circ] \\
= & ((\lambda n.M^\circ)[l\mapsto\mu l.\lambda\vec{y}.L^\circ])((\mu n.\lambda\vec{x}.N^\circ)[l\mapsto\mu l.\lambda\vec{y}.L^\circ]) \\
= & (((\lambda l.M^\circ)(\mu l.\lambda\vec{y}.L^\circ))\,(((\lambda l.N^\circ)(\mu l.\lambda\vec{y}.L^\circ)) \\
= & (M\{l\langle\vec{y}\rangle{\Leftarrow}L\}\{n\langle\vec{x}\rangle{\Leftarrow}N\{l\langle\vec{y}\rangle{\Leftarrow}L\}\})^\circ
\end{array}
$$

□

For the non-recursive fragment of CPS calculus, one can simplify the translation to λ-calculus, not requiring the fixpoint combinator.

2.6.2 Definition

$$
\begin{array}{rcl}
k\langle x_1\ldots x_n\rangle^\circ & = & k(x_1,\ldots,x_n) \\
M\{n\langle x_1\ldots x_n\rangle{=}N\}^\circ & = & (\lambda n.M^\circ)(\lambda(x_1,\ldots,x_n).N^\circ)
\end{array}
$$

2.6.3 Remark Alternatively, we could use simply-typed λ-calculus by uncurrying the translation.

$$\begin{array}{rcl} k\langle x_1 \dots x_n\rangle^\circ & = & k x_1 \dots x_n \\ M\{n\langle x_1 \dots x_n\rangle{=}N\}^\circ & = & (\lambda n.M^\circ)(\lambda x_1 \dots x_n.N^\circ) \end{array}$$

2.6.4 Definition CPS calculus can be translated into the π-calculus as follows. (Note that continuation binding is essentially Sangiorgi's "local environment" idiom for the π-calculus [PS96].)

$$\begin{array}{rcl} k\langle x_1 \dots x_i\rangle^\bullet & = & \overline{k}\langle x_1 \dots x_i\rangle \\ M\{n\langle x_1 \dots x_i\rangle{=}N\}^\bullet & = & (\nu n)(M^\bullet \mid !n(x_1 \dots x_i).N^\bullet) \end{array}$$

2.6.5 Remark In the most general π-calculus, the CPS laws are not sound. CPS gives rise to π-calculus terms of a very restricted from: there are constraints both one the occurrence of names and on the shape of terms.

On the one hand, all names are used as continuations. A precise formulation of what constitutes a continuation type discipline in the π-calculus seems to be an open problem, although a necessary (but not sufficient) constraint could be given in terms of the input/output type discipline developed in [PS96]. All names are "write-only" in that a name that has been received may be used by the receiver only for output, but not for input (it may also be passed as an argument).

Furthermore, all names are ω-receptive in the sense of [San97].

The terms in the image of the translation from the CPS calculus to the π-calculus are of a the following form, in which restriction, parallel composition, replication and input prefix occur only in an idiom and never by themselves. All outputs are asynchronous.

$$P, Q ::= \overline{k}\langle x_1 \dots x_n\rangle \mid (\nu q)(P \mid !q(y_1 \dots y_m).Q)$$

This restricted form automatically rules out usages of names like $n(k).k(x)$.

A variant of the π-calculus that is more permissive than CPS, but still restritive enough to be "well-behaved" with regard to it is the calculus Π_a^-.

2.6.6 Definition The fragment Π_a^- of the π-calculus is given by the following BNF:

$$P \quad ::= \quad P|P \mid a(\vec{x}).P \mid !a(\vec{x}).P \mid (\nu a)P \mid \overline{a}\langle\vec{x}\rangle$$

where in $a(x_1 \dots x_n).P$ none of the x_i appears in input position (as in $x_i(\vec{y}).Q$) within P.

We consider terms of Π_a^- equivalent up to *barbed congruence* [ACS96], written $\approx_{\mathsf{bc}}$.

Hence received names can be only passed around or used for output. This together with the fact that output is asynchronous, appears to be enough to ensure a sufficiently continuation-like behaviour. More formally, we report the following result, due to Massimo Merro:

2.6.7 Proposition The translation given in definition 2.6.4 above from CPS calculus to $\Pi_a^-/\approx_{\mathsf{bc}}$ is sound. For CPS terms M and N, $M = N$ implies $M^\bullet \approx_{\mathsf{bc}} N^\bullet$.

Another translation of CPS calculus is given by a "continuation-grabbing style" transformation similar to that in [Sab96], which transforms CPS terms back into idealised NJ-SML.

2.6.8 Definition The Continuation Grabbing Transform $(\_)^\ddagger$ is defined as follows.

$$\begin{aligned} M\{n\langle x_1 \ldots x_j\rangle{=}N\}^\ddagger &\stackrel{\text{def}}{=} (\lambda(x_1,\ldots,x_j).N^\ddagger)(\mathtt{callcc}\,(\lambda n.M^\ddagger)) \\ k\langle x_1 \ldots x_j\rangle^\ddagger &\stackrel{\text{def}}{=} \mathtt{throw}\,k\,(x_1,\ldots,x_j) \end{aligned}$$

On types, this translation is given by the identity.

2.6.9 Remark If $\Gamma \vdash M$, then $\Gamma \vdash M^\ddagger : \beta$, where β is a fresh type variable.

Proof For terms of the form $k\langle x_1 \ldots x_j\rangle$, this is trivial. Consider $\Gamma \vdash M\{n\langle\vec{x}\rangle{=}N\}$. Then

$$\Gamma, n : \neg\vec{\tau} \vdash M \quad \text{and} \quad \Gamma, \vec{x} : \vec{\tau} \vdash N$$

By the induction hypothesis,

$$\Gamma, n : \neg\vec{\tau} \vdash M^\ddagger : \beta \quad \text{and} \quad \Gamma, \vec{x} : \vec{\tau} \vdash N^\ddagger : \beta$$

Instantiating β to $\vec{\tau}$ we have $\Gamma, n : \neg\vec{\tau} \vdash M^\ddagger : \vec{\tau}$ and therefore $\Gamma \vdash \lambda\vec{x}.M^\ddagger : \neg\vec{\tau} \to \vec{\tau}$. Because $\vdash \mathtt{callcc} : (\neg\alpha \to \alpha) \to \alpha$, we have $\Gamma \vdash \mathtt{callcc}(\lambda\vec{x}.M^\ddagger) : \vec{\tau}$. Now $\Gamma \vdash \lambda\vec{x}.N^\ddagger : \vec{\tau} \to \beta$, so

$$\Gamma \vdash (\lambda\vec{x}.N^\ddagger)(\mathtt{callcc}(\lambda\vec{x}.M^\ddagger)) : \beta.$$

$\square$

2.7 Idioms and jargon for the CPS calculus

We sometimes use jargon, mainly from programming language theory, together with some loan words from the π-calculus literature, for talking about CPS. This is not done for obfuscation, but to make the presentation more intuitive for the reader, who we assume is likely to be familiar with most of these terms. As a preparation for chapter 3, we give a brief discussion. The purpose is not to give terminology for different parts of a CPS term (hardly necessary, since the calculus is so simple), but to certain idioms and points of view on terms.

Like the π-calculus, the CPS calculus is a "name-passing" calculus: the only entities that may be substituted for variables are other variables (also called "names"):

$$n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle{=}N\} \quad = \quad N[\vec{z} \mapsto \vec{y}]$$

Because of this name-paaing, a notational shortcut we shall perpetrate is the simulation of substitution by α-conversion. We rename the bound parameters before contracting a redex.

$$\begin{array}{ll} & a\langle b\rangle\{a\langle x\rangle{=}x\langle c\rangle\} \\ = & a\langle b\rangle\{a\langle b\rangle{=}b\langle c\rangle\} \\ = & b\langle c\rangle \end{array}$$

This saves us from having to write substitutions.

$$\begin{array}{ll} & a\langle b\rangle\{a\langle x\rangle{=}x\langle c\rangle\} \\ = & x\langle c\rangle[x \mapsto b] \\ = & b\langle c\rangle \end{array}$$

A characteristic feature of (reductions in) the CPS calculus is a kind of leapfrogging of bindings like this:

$$\begin{array}{ll} & k\langle f\rangle\{f\langle xk\rangle{=}M\}\{k\langle f\rangle{=}N\{n\langle x\rangle{=}f\langle xk\rangle\}\} \\ = & N\{n\langle x\rangle{=}f\langle xk\rangle\}\{f\langle xk\rangle{=}M\} \\ = & N\{n\langle x\rangle{=}M\} \end{array}$$

The same term may mean quite different things, depending on what we regard as the current continuation. (One of the themes of the the categorical framework is a development of this fact: one of the most basic operations is precisely this switch of current continuation.) In the CPS calculus, there is no notion of a *current* continuation as such; all continuations are equal. Nevertheless, when reading a term in a structured fashion, it is often essential to single out one of possibly many names as the current continuation. We consider two ways of doing this. One is always to look at judgements rather than raw terms; the other, more informal and ad hoc, is to use the same name, typically k, for the current continuation everywhere.

The translation of λ-terms gives the spirit of (nearly all) CPS transforms.

$$\overline{\lambda x.M} = k\langle f\rangle\{f\langle xk\rangle{=}\overline{M}\}$$

Here we call f a *pivot*. This concept seems absent from the usual CPS terminology, probably because CPS is not normally presented as name-passing. So we borrowed it from Pierce and Sangiorgi [PS93]; they write about the translation of a λ-calculus term MN into the π-calculus:

> *The core of the protocol [...] is the action on an internal channel v, by which the abstraction M comes to know its arguments. We call v a* pivot*. (In the lazy λ-calculus encoding, the role of the pivot names was played by the argument port names.)*

Warning: Milner's call-by-value π-calculus transformations have an additional level of indirection not present in CPS. So our usage of pivot is not completely the same as that of Sangiorgi and Pierce.

[Ing61] defines

> *A* thunk *is a piece of coding that provides an address. When executed, it leaves in some standard location (memory, accumulator, or index register, for example) the address of the variable with which it is associated.*

In the present setting, a thunk is a term of the form $k\langle q\rangle\{q\langle p\rangle{=}M\}$. $k\langle q\rangle\{q\langle p\rangle{=}M\}$ returns to its current continuation a "private" name q along which it is ready to receive a continuation p for M; M may then evaluate and return a result to p.

Complementary to thunking, a forcing is a jump with the current continuation as the actual parameter.

Chapter 3

CPS transforms

The main purpose of this chapter is to review, and present in a unified notation, the various CPS transforms that have appeared in the literature (the seminal papers are [Plo75] and [DHM91] for `callcc`; a unified account is in [DH94]. See also [Fil96].)

There is a large literature on the typing of CPS transforms, beginning with [MW85], later with [Gri90] and in particular Murthy, e.g.. [Mur91]

The paradigmatic language that we consider as the source language for CPS transforms is a simply-typed λ-calculus, usually (but not in all cases) augmented by the control operators `callcc` and `throw`. The BNF for raw terms is this:

$$M \quad ::= x \mid \lambda x.M \mid MM \mid \mathtt{callcc}\, M \mid \mathtt{throw}\, M$$

We call this language λ+`callcc`; it is essentially the same as that in [DHM91].

To prevent misunderstanding, one should perhaps emphasise that this calculus is a (standard) idealisation of call-by-value programming languages like Scheme or ML and semantically very different from the simply-typed λ-calculus whose models are Cartesian closed categories, see [LS86].

We do not give an operational semantics here, as we regard the CPS transforms as its proper semantics, but the intended meaning of the calculus is that it should be a call-by-value calculus in the sense of [Plo75], where β and η laws appy only in a restricted sense.

$$\begin{array}{rcll} (\lambda x.M)V & = & M[x \mapsto V] & (\beta_V) \\ \lambda y.Vy & = & V & (\eta_V) \\ & & \text{where } y \text{ is not free in } M & \end{array}$$

Here V ranges over values, i.e. terms of the form $V ::= x \mid \lambda x.M$.

The typing for first-class continuations in Standard ML of New Jersey in [DHM91] is given by that of simply-typed λ-calculus and the two rules for the continuation primitives `callcc` and `throw`.

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \qquad \frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\Gamma \vdash M : \neg\tau \to \tau}{\Gamma \vdash \mathtt{callcc}\ M : \tau} \qquad \frac{\Gamma \vdash M : \neg\tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash \mathtt{throw}\ M\ N\ : \sigma}$$

3.1 A survey of CPS transforms

From our point of view, the canonical CPS transform is the one from [DHM91].

3.1.1 Definition The call-by-value transform $(\!|\_|\!)$ for λ+$\mathtt{callcc}$ is defined as follows.

$$\begin{array}{rcl}
(\!|x|\!)(k) & = & k\langle x\rangle \\
(\!|\lambda x.M|\!)(k) & = & k\langle f\rangle\{f\langle xh\rangle \Leftarrow (\!|M|\!)(h)\} \\
(\!|MN|\!)(k) & = & (\!|M|\!)(m)\{m\langle f\rangle \Leftarrow (\!|N|\!)(n)\{n\langle a\rangle \Leftarrow f\langle ak\rangle\}\} \\
(\!|\mathtt{throw}\ M\ N|\!)(k) & = & (\!|M|\!)(m)\{m\langle n\rangle \Leftarrow (\!|N|\!)(n)\} \\
(\!|\mathtt{callcc}\ M|\!)(k) & = & (\!|M|\!)(m)\{m\langle f\rangle \Leftarrow f\langle kk\rangle\}
\end{array}$$

$$\begin{array}{rcl}
(\!|\neg\tau|\!) & = & \neg(\!|\tau|\!) \\
(\!|\sigma \to \tau|\!) & = & \neg((\!|\sigma|\!)\ \neg(\!|\tau|\!)) \\
(\!|x_1 : \tau_1, \ldots x_n : \tau_n|\!) & = & (\!|\tau_1|\!), \ldots, (\!|\tau_n|\!)
\end{array}$$

This is almost the same as the transform used by Appel in his ML compiler [App92]. We recall here what he calls the naive version, which does not deal with exception handlers.

3.1.2 Definition We write $x \mapsto \ldots$ for meta-level abstraction, i.e. ordinary function abstraction, not construction of a λ-term, and $c(x)$ for the corresponding meta-level application. The following is Appel's CPS transform (with the naive version of `callcc`) [App92, ch. 5].

$$\begin{array}{rcl}
\mathcal{A}(\!|-|\!) & : & \mathit{Lambda} \to (\mathit{Name} \to \mathit{CPS}) \to \mathit{CPS} \\
\mathcal{A}(\!|x|\!)(c) & = & c(x) \\
\mathcal{A}(\!|\lambda x.M|\!)(c) & = & c(f)\{f\langle xk\rangle = \mathcal{A}(\!|M|\!)(z \mapsto k\langle z\rangle)\} \\
\mathcal{A}(\!|MN|\!)(c) & = & \mathcal{A}(\!|M|\!)(m \mapsto \mathcal{A}(\!|N|\!)(n \mapsto m\langle nk\rangle))\{k\langle x\rangle = c(x)\} \\
\mathcal{A}(\!|\mathtt{callcc}\ M|\!)(c) & = & \mathcal{A}(\!|M|\!)(m \mapsto m\langle kk\rangle)\{k\langle x\rangle = c(x)\} \\
\mathcal{A}(\!|\mathtt{throw}\ M|\!)(c) & = & \mathcal{A}(\!|M|\!)(m \mapsto c(f)\{f\langle xh\rangle = m\langle x\rangle\})
\end{array}$$

Appel's transform performs a certain amount of administrative reductions. Up to provable equality of the CPS calculus, the transform $\mathcal{A}(\!|\_|\!)$ is the same as $(\!|\_|\!)$.

3.1.3 Proposition

$$\mathcal{A}(\!|M|\!)(c)) = (\!|M|\!)(k)\{k\langle x\rangle{=}c(x)\}$$

Hence $\mathcal{A}(\!|M|\!)(z \mapsto k\langle z\rangle) = (\!|M|\!)(k)$

The call-by-value transform satisfies only the β_V law, not the full β law.

Plotkin's call-by-name transform satisfies the full β-law. However, despite being traditionally called "call-by-name", the transform does not satisfy the full η-law.

This is because it is "lazy" in the sense that λ-abstraction delays the evaluation of the body (sometimes called "protecting by a λ [Plo75])". We could qualify "call-by-name" with "lazy" to distinguish this transform from alternatives, not afflicted by laziness, that satisfy the full η law. Unfortunately, the term "lazy" is sometimes used to mean call-by-need in the sense of memoisation of arguments.

3.1.4 Definition The Plotkin call-by-name CPS transform [Plo75] is defined as follows.

$$\begin{array}{rcl}
\mathcal{L}(\!|x|\!)(k) & = & x\langle k\rangle \\
\mathcal{L}(\!|\lambda x.M|\!)(k) & = & k\langle f\rangle\{f\langle xh\rangle{\Leftarrow}\mathcal{L}(\!|M|\!)(h)\} \\
\mathcal{L}(\!|MN|\!)(k) & = & \mathcal{L}(\!|M|\!)(m)\{m\langle f\rangle{\Leftarrow}f\langle nk\rangle\{n\langle h\rangle{\Leftarrow}\mathcal{L}(\!|N|\!)(h)\}\}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{L}(\!|\sigma \to \tau|\!) & = & \neg(\neg\neg\mathcal{L}(\!|\sigma|\!), \mathcal{L}(\!|\tau|\!)) \\
\mathcal{L}(\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!) & = & \neg\neg\mathcal{L}(\!|\tau_1|\!), \ldots, \neg\neg\mathcal{L}(\!|\tau_n|\!)
\end{array}$$

3.1.5 Remark Note that the argument N in an application MN is not forced, but only located at n. While this may look simpler than the corresponding clause for call-by-value, we can nonetheless regard it as a special case of it.

$$\begin{array}{rcl}
\mathcal{L}(\!|MN|\!)(k) & = & \mathcal{L}(\!|M|\!)(m)\{m\langle f\rangle{\Leftarrow}f\langle nk\rangle\{n\langle h\rangle{\Leftarrow}\mathcal{L}(\!|N|\!)(h)\}\} \\
& = & \mathcal{L}(\!|M|\!)(m)\{m\langle f\rangle{\Leftarrow}q\langle n\rangle\{n\langle h\rangle{\Leftarrow}\mathcal{L}(\!|N|\!)(h)\}\{q\langle n\rangle{\Leftarrow}f\langle nk\rangle\}\}
\end{array}$$

Written this way, the application follows the same pattern as for call-by-value,

$$\mathcal{L}(\!|M|\!)(m)\{m\langle f\rangle{\Leftarrow}\ldots\{q\langle n\rangle{\Leftarrow}f\langle nk\rangle\}\}$$

except that the term in the argument position, represented by $\ldots$, is not just $\mathcal{L}(\!|N|\!)(h)$, but $\mathcal{L}(\!|N|\!)(h)$ wrapped into a thunk:

$$q\langle n\rangle\{n\langle h\rangle{\Leftarrow}\mathcal{L}(\!|N|\!)(h)\}$$

Note that the whole computation is wrapped into the thunk, so that $\mathcal{L}(\!|N|\!)(h)$ is not evaluated at this point. Alternatively, one *could* evaluate it, and wrap the result (if any) into a thunk:

$$\mathcal{L}(\!|N|\!)(h)\{h\langle x\rangle \Leftarrow q\langle n\rangle\{n\langle h\rangle \Leftarrow h\langle x\rangle\}\}$$

This would give a variant CPS transform $\mathcal{L}'(\!|\_|\!)$, agreeing with $\mathcal{L}(\!|\_|\!)$ except the clause for application, in which the result, not the computation, is thunked:

$$\begin{array}{ll} & \mathcal{L}'(\!|MN|\!)(k) \\ = & \mathcal{L}'(\!|M|\!)(m)\{m\langle f\rangle \Leftarrow \mathcal{L}'(\!|N|\!)(h)\{h\langle x\rangle \Leftarrow q\langle n\rangle\{n\langle h\rangle \Leftarrow h\langle x\rangle\}\}\{q\langle n\rangle \Leftarrow f\langle nk\rangle\}\} \\ = & \mathcal{L}'(\!|M|\!)(m)\{m\langle f\rangle \Leftarrow \mathcal{L}'(\!|N|\!)(h)\{h\langle x\rangle \Leftarrow f\langle nk\rangle\{n\langle h\rangle \Leftarrow h\langle x\rangle\}\}\} \end{array}$$

Hence, to reiterate a point made in [DH94], the typing of call-by-name does not imply call-by-name behaviour.

The Plotkin call-by-name transform can be seen as a modification of the call-by-value transform, given by delaying the argument in an application and forcing variables. According to this view, it is merely one of a number of possible variations arising from different choices about the point in the computation at which arguments are to be evaluated. Two other such variations, called the Reynolds and modified Reynolds CPS transform, are presented in [DH94].

3.1.6 Definition The Reynolds call-by-value CPS transform [DH94] is defined as follows.

$$\begin{array}{rcl} \mathcal{R}(\!|x|\!)(k) & = & x\langle k\rangle \\ \mathcal{R}(\!|\lambda x.M|\!)(k) & = & k\langle f\rangle\{f\langle nh\rangle = n\langle h'\rangle\{h'\langle a\rangle = \mathcal{R}(\!|M|\!)(h)\{x\langle h''\rangle = h''\langle a\rangle\}\}\} \\ \mathcal{R}(\!|MN|\!)(k) & = & \mathcal{R}(\!|M|\!)(m)\{m\langle f\rangle = f\langle nk\rangle\{n\langle h\rangle = \mathcal{R}(\!|N|\!)(h)\}\} \end{array}$$

3.1.7 Definition The modified Reynolds call-by-value CPS transform [DH94] is defined as follows.

$$\begin{array}{rcl} \mathcal{R}'(\!|x|\!)(k) & = & k\langle x\rangle \\ \mathcal{R}'(\!|\lambda x.M|\!)(k) & = & k\langle f\rangle\{f\langle nh\rangle = n\langle h'\rangle\{h'\langle x\rangle = \mathcal{R}'(\!|M|\!)(h)\}\} \\ \mathcal{R}'(\!|MN|\!)(k) & = & \mathcal{R}'(\!|M|\!)(m)\{m\langle f\rangle = f\langle nk\rangle\{n\langle h\rangle = \mathcal{R}'(\!|N|\!)(h)\}\} \end{array}$$

The reason why the call-by-name CPS transform does not satisfy the η-law is, roughly speaking, that λ "protects" an application from being evaluated in the evaluation strategy codified by it. Hence in $\lambda x.MNx$, the application is protected, whereas in MN, M is evaluated.

There are two possible ways one could try to address this discrepancy: either one could try to reduce under the λ in the case of $\lambda x.MNx$; or one could avoid evaluating the M in MN.

Murthy defines what he calls a "truly" call-by-name CPS transform, which can be seen as reducing under λ.

3.1.8 Definition The Murthy call-by-name CPS transform [Mur92] is defined as follows.

$$\begin{array}{rcl} \mathcal{M}(\!|x|\!)(k) & = & x\langle k\rangle \\ \mathcal{M}(\!|\lambda x.M|\!)(k) & = & \mathcal{M}(\!|M|\!)(m)\{m\langle b\rangle{=}k\langle f\rangle\{f\langle ak\rangle{=}k\langle b\rangle\}\}\{x\langle h\rangle{=}k\langle f\rangle\{f\langle ak\rangle{=}a\langle h\rangle\}\} \\ \mathcal{M}(\!|MN|\!)(k) & = & \mathcal{M}(\!|M|\!)(m)\{m\langle f\rangle{=}f\langle nk\rangle\{n\langle h\rangle{=}\mathcal{M}(\!|N|\!)(h)\}\} \end{array}$$

Unfortunately, to be as well-behaved as claimed, this transform requires a different notion of equality than provability by the CPS axioms. So in the present setting, we can not really make much use of it. What seems fascinating about this transform, though, is that returning the result and accessing the argument are in some sense two separate processes. Only when the argument is needed is a request sent to the calling context. This could mean that Murthy's transform is especially suitable for a concurrent scenario. The appropriate notion of equality for this transform may be the observational congruence $\approx$ from definition 2.3.2.

The other, in some sense dual, possibility to make $\lambda x.MNx$ indistinguishable from MN, we have indicated, is to not to force the evaluation of M in an application MN. This is essentially what the uncurrying call-by-name CPS transform does.

3.1.9 Definition The uncurrying call-by-name CPS transform is defined as follows.

$$\begin{array}{rcl} \mathcal{N}(\!|x|\!)(k) & = & k\langle x\rangle \\ \mathcal{N}(\!|\lambda x.M|\!) & = & k\langle f\rangle\{f\langle x\vec{y}h\rangle{\Leftarrow}\mathcal{N}(\!|M|\!)(m)\{m\langle g\rangle{\Leftarrow}g\langle\vec{y}h\rangle\}\} \\ \mathcal{N}(\!|MN|\!) & = & \mathcal{N}(\!|M|\!)(m)\{m\langle f\rangle{\Leftarrow}\mathcal{N}(\!|N|\!)(n)\{n\langle a\rangle{\Leftarrow}k\langle g\rangle\{g\langle\vec{y}k\rangle{\Leftarrow}f\langle a\vec{y}k\rangle\}\}\} \end{array}$$

$$\begin{array}{rcl} \mathcal{N}(\!|\tau_1 \to \cdots \to \tau_n \to b|\!) & = & \neg(\mathcal{N}(\!|\tau_1|\!),\ldots,\mathcal{N}(\!|\tau_n|\!),\neg b) \\ \mathcal{N}(\!|x_1:\tau_1,\ldots,x_n:\tau_n|\!) & = & \mathcal{N}(\!|\tau_1|\!),\ldots,\mathcal{N}(\!|\tau_n|\!) \end{array}$$

This transform is related to Filinski's call-by-name transform in [Fil96] in that it doubly negates base types. In its name-passing presentation it appears to be new. There may be a connection to graph reduction here, inasmuch as an application with the result not being of base type does essentially nothing, other than consing the given argument onto the argument list. A comparison with the G-machine constructing an application graph [FH88] seems possible here.

3.2 A simplified notation for non-recursive CPS

For the non-recursive CPS calculus, we can present the CPS transforms in a shorthand style in which the CPS transform ($\overline{M}$, $\underline{M}$ and $\underline{\underline{M}}$) of a term contains a free variable k for the current continuation. (This notational trick appears to be due to Phil Wadler; see [SW96].)

This may seem utterly confusing at first, to the extent that just about everything appears to be called k. But as the CPS calculus is a low-level name-passing calculus, it is virtually intractable without some technique for simplifying notation. Given that CPS terms admit a very "imperative" reading, one could add a more computational justification for always calling the current continuation k, in that we could think of k as always being the same register (current continuation pointer), but at different times during the computation. In that sense, the fact that the variable k can be recycled endlessly reflects the fact that the non-recursive CPS calculus can only express *forward* jumps. One can destructively update the current continuation pointer, as one can never jump back to it.

3.2.1 Definition The call-by-value with `callcc` CPS transform in the shorthand notation $\overline{(\_)}$ is defined as follows.

$$\begin{array}{rcl}
\overline{x} & = & k\langle x\rangle \\
\overline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\overline{M}\} \\
\overline{MN} & = & \overline{M}\{k\langle f\rangle{=}\overline{N}\{k\langle a\rangle{=}f\langle ak\rangle\}\} \\
\overline{\texttt{callcc}\ M} & = & \overline{M}\{k\langle f\rangle{=}f\langle kk\rangle\} \\
\overline{\texttt{throw}\ M\ N} & = & \overline{M}\{k\langle k\rangle{=}\overline{N}\} \\
\overline{(M,N)} & = & \overline{M}\{k\langle m\rangle{=}\overline{N}\{k\langle n\rangle{=}k\langle mn\rangle\}\} \\
\overline{\sigma\rightarrow\tau} & = & \neg(\overline{\sigma}\neg\overline{\tau})
\end{array}$$

For the Plotkin call-by-name semantics, the semantics of the control operators is somewhat tentative, as their intended meaning is not as clear as for call-by-value. The absence of implementations of call-by-name languages with control operators makes it hard to give pragmatic evidence as to what the definition of `callcc` should look like. Nonetheless, if we view the Plotkin call-by-name CPS transform as esssentially the same as call-by-name modulo thunking of function arguments, we can argue that call-by-name `callcc` should be as in call-by-value, with the only modification that the current continuation, once it has been seized, is wrapped into a thunk. This allows us to keep the same typing rule for `callcc`.

3.2.2 Definition The Plotkin call-by-name CPS transform in the shorthand notation $\underline{(\_)}$ is defined as follows.

$$
\begin{array}{rcl}
\underline{x} & = & x\langle k\rangle \\
\underline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\underline{M}\} \\
\underline{MN} & = & \underline{M}\{k\langle f\rangle{=}f\langle nk\rangle\{n\langle k\rangle{=}\underline{N}\}\} \\
\underline{\mathtt{callcc}\ M} & = & \underline{M}\{k\langle f\rangle{=}f\langle gk\rangle\{g\langle p\rangle{=}p\langle k\rangle\}\} \\
\underline{\mathtt{throw}\ M\ N} & = & \underline{M}\{k\langle k\rangle{=}\underline{N}\} \\
\underline{\sigma \to \tau} & = & \neg(\neg\neg\underline{\sigma}\,\neg\underline{\tau})
\end{array}
$$

3.2.3 Definition The uncurrying call-by-name CPS transform in the shorthand notation $\underline{\underline{(\_)}}$ is defined as follows.

$$
\begin{array}{rcl}
\underline{\underline{x}} & = & k\langle x\rangle \\
\underline{\underline{\lambda x.M}} & = & k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\} \\
\underline{\underline{MN}} & = & \underline{\underline{M}}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle a\rangle{=}k\langle g\rangle\{g\langle\vec{y}k\rangle{=}f\langle a\vec{y}k\rangle\}\}\} \\
\underline{\underline{\mathtt{callcc}\ M}} & = & k\langle f\rangle\{f\langle\vec{y}h\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle g\vec{y}h\rangle\{g\langle q\rangle{=}q\langle\vec{y}h\rangle\}\}\} \\
\underline{\underline{\mathtt{throw}\ M\ N}} & = & k\langle f\rangle\{f\langle\vec{y}h\rangle{=}\underline{\underline{M}}\{k\langle k\rangle{=}\underline{\underline{N}}\}\}
\end{array}
$$

$$
\underline{\underline{\tau_1 \to \ldots \to \tau_n \to b}} \quad = \quad \neg(\underline{\underline{\tau_1}} \ldots \underline{\underline{\tau_n}}\,\neg b)
$$

One difference to CPS transforms using the λ-calculus as the target language is that there is no way to define the abort operator by using the $\lambda x.x$ as an aborting continuation (as in [FFKD86]). But we can define Danvy and Filinski's meta-continuation passing style [DF92].

3.2.4 Definition For λ-calculus together with `callcc`, `throw`, the abort operator $\mathcal{A}$ and the control delimiter $\#$, i.e. the following language

$$
M \quad ::= x \mid \lambda x.M \mid MM \mid \mathtt{callcc}\,M \mid \mathtt{throw}\,M \mid \mathcal{A}\,M \mid \#\,M
$$

the Danvy/Filinski metacontinuation transform $\overline{\overline{(\_)}}$ is given as follows.

$$
\begin{array}{rcl}
\overline{\overline{x}} & = & k\langle xc\rangle \\
\overline{\overline{\lambda x.M}} & = & k\langle fc\rangle\{f\langle xkc\rangle{=}\overline{\overline{M}}\}
\end{array}
$$

$$
\begin{aligned}
\overline{\overline{MN}} &= \overline{\overline{M}}\{k\langle fc\rangle{=}\overline{\overline{N}}\{k\langle ac\rangle{=}f\langle akc\rangle\}\} \\
\overline{\overline{\texttt{callcc}\ M}} &= \overline{\overline{M}}\{k\langle fc\rangle{=}f\langle kkc\rangle\} \\
\overline{\overline{\texttt{throw}\ M\ N}} &= \overline{\overline{M}}\{k\langle kc\rangle{=}\overline{\overline{N}}\} \\
\overline{\overline{\#\,M}} &= \overline{\overline{M}}\{k\langle xc\rangle{=}c\langle x\rangle\}\{c\langle x\rangle{=}k\langle xc\rangle\} \\
\overline{\overline{\mathcal{A}\,M}} &= \overline{\overline{M}}\{k\langle xc\rangle{=}c\langle x\rangle\}
\end{aligned}
$$

The semantics of the composable continuation construct $\mathcal{S}$ is given as follows.

$$\overline{\overline{\mathcal{S}\,M}} = \overline{\overline{M}}\{k\langle fc\rangle{=}f\langle gkc\rangle\{k\langle xc\rangle{=}c\langle x\rangle\}\{g\langle yk'c'\rangle{=}k\langle yc\rangle\{c\langle w\rangle{=}k'\langle yc'\rangle\}\}\}$$

3.3 Soundness of the uncurrying call-by-name CPS transform

The transform $\underline{\underline{(\_)}}$ is genuinely call-by-name in that it satisfies both the unrestricted β- and η-laws. We show this for the non-recursive CPS calculus. Using the shorthand with the current continuation always being called k, we can keep track of what is happening without being swamped by too many distinct variables.

The crucial property of the uncurrying call-by-name CPS transform is that the transform of all λ-terms is thunkable. For the call-by-value CPS transform, by contrast, this would hold only for the transform of *values*.

3.3.1 Lemma Let L be a λ-term. Then its uncurrying call-by-name transform $\underline{\underline{L}}$ satisfies:

$$\underline{\underline{L}} = k\langle q\rangle\{q\langle\vec{z}\rangle{=}\underline{\underline{L}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\}$$

Proof By induction on L. Let $L = x$.

$$
\begin{aligned}
& k\langle q\rangle\{q\langle\vec{z}\rangle{=}\underline{\underline{x}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= {} & k\langle q\rangle\{q\langle\vec{z}\rangle{=}k\langle x\rangle\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= {} & k\langle q\rangle\{q\langle\vec{z}\rangle{=}x\langle\vec{z}\rangle\} \\
= {} & k\langle x\rangle \\
= {} & \underline{\underline{x}}
\end{aligned}
$$

Suppose $L = \lambda x.M$ and let the induction hypothesis hold for M.

$$
\begin{aligned}
& k\langle q\rangle\{q\langle\vec{z}\rangle{=}\underline{\underline{\lambda x.M}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= {} & k\langle q\rangle\{q\langle\vec{z}\rangle{=}k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= {} & k\langle q\rangle\{q\langle\vec{z}\rangle{=}f\langle\vec{z}\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}\} \\
= {} & k\langle q\rangle\{q\langle x\ ysk\rangle{=}f\langle x\vec{y}k\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}\}
\end{aligned}
$$

$$
\begin{aligned}
&= k\langle q\rangle\{q\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\} \\
&= \underline{\underline{\lambda x.M}}
\end{aligned}
$$

Suppose $L = MN$ with M and N satisfying the induction hypothesis.

$$
\begin{aligned}
& k\langle q\rangle\{q\langle\vec{z}\rangle{=}\underline{\underline{MN}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= \;& k\langle q\rangle\{q\langle\vec{z}\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle x\rangle{=}k\langle g\rangle\{g\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\}\}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= \;& k\langle q\rangle\{q\langle\vec{z}\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle x\rangle{=}g\langle\vec{z}\rangle\{g\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\}\}\} \\
= \;& k\langle q\rangle\{q\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle x\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& k\langle q\rangle\{q\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}k\langle x\rangle\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\}\{k\langle x\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& k\langle q\rangle\{q\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\}\}\} \\
= \;& k\langle q\rangle\{q\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\}\}\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= \;& k\langle x\rangle\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\}\{k\langle x\rangle{=}k\langle q\rangle\{q\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& \underline{\underline{N}}\{k\langle x\rangle{=}k\langle q\rangle\{q\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& \underline{\underline{N}}\{k\langle x\rangle{=}k\langle q\rangle\{q\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\}\}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle x\rangle{=}k\langle q\rangle\{q\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& \underline{\underline{M}}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle x\rangle{=}k\langle q\rangle\{q\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\}\} \\
= \;& \underline{\underline{MN}}
\end{aligned}
$$

□

As a preparation for the soundness of β-reduction, we need to establish how, under the transform $\underline{\underline{(\_)}}$, substitution in the source language, λ-calculus, relates to a binding in the target, CPS calculus. Because of lemma 3.3.1, we have two equivalent views on what a substitution $\underline{\underline{M[x \mapsto N]}}$ means: we can see it as M, expecting the argument x, becoming the current continuation for N; or we can see it as M having access, via the pointer x, to the resource N, expecting its current continuation as its argument. In that sense, the uncurrying call-by-name transform provides a "pure" semantics for simply-typed λ-calculus: the denotation (transform) of everything is as good as a value.

3.3.2 Lemma

$$\underline{\underline{M[x \mapsto N]}} = \underline{\underline{N}}\{k\langle x\rangle{=}\underline{\underline{M}}\}$$

Proof By lemma 3.3.1, this is equivalent to

$$
\begin{aligned}
\underline{\underline{M[x \mapsto N]}} &= k\langle x\rangle\{x\langle p\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\}\{k\langle x\rangle{=}\underline{\underline{M}}\} \\
&= \underline{\underline{M}}\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\}
\end{aligned}
$$

If $M = x$, then

$$\underline{\underline{M}}\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} = k\langle x\rangle\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} = \underline{\underline{N}}$$

If $M = y \neq x$, then

$$\underline{\underline{M}}\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} = k\langle y\rangle\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} = \underline{\underline{y}}$$

The remaining cases follow by induction using the fact that the binding for x distributes — except for the case when the scope of x has a hole because x is λ-bound:

$$\begin{aligned}
& \underline{\underline{(\lambda x.M)[x \mapsto N]}} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}M\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}\{x\langle\vec{z}\rangle{=}\underline{\underline{N}}\{k\langle p\rangle{=}p\langle\vec{z}\rangle\}\} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}M\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}
\end{aligned}$$

because x is not free in $k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}M\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}$. □

The η-law is sound for $\underline{\underline{(\_)}}$ just as η_V is sound for $\overline{(\_)}$.

3.3.3 Proposition (η for $\underline{\underline{(\_)}}$)

$$\underline{\underline{\lambda x.Mx}} = \underline{\underline{M}}$$

where $x \notin \mathsf{FV}(M)$.

Proof

$$\begin{aligned}
& \underline{\underline{\lambda x.Mx}} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{Mx}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}k\langle g\rangle\{g\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}\} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle f\rangle{=}f\langle x\vec{y}k\rangle\}\} \\
= \;& \underline{\underline{M}} \qquad \text{by lemma 3.3.1}
\end{aligned}$$

□

3.3.4 Proposition (β for $\underline{\underline{(\_)}}$)

$$\underline{\underline{(\lambda x.M)N}} = \underline{\underline{M[x \mapsto N]}}$$

Proof

$$\begin{aligned}
& \underline{\underline{(\lambda x.M)N}} \\
= \;& k\langle f\rangle\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\}\{k\langle f\rangle{=}\underline{\underline{N}}\{k\langle x\rangle{=}k\langle g\rangle\{g\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\}\}
\end{aligned}$$

$$
\begin{aligned}
&= \underline{\underline{N}}\{k\langle x\rangle{=}k\langle g\rangle\{g\langle\vec{y}k\rangle{=}f\langle x\vec{y}k\rangle\}\}\{f\langle x\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}g\langle\vec{y}k\rangle\}\} \\
&= \underline{\underline{N}}\{k\langle x\rangle{=}k\langle g\rangle\{g\langle\vec{y}k\rangle{=}\underline{\underline{M}}\{k\langle g\rangle{=}k\langle g\rangle\}\}\} \qquad \text{by lemma 3.3.1} \\
&= \underline{\underline{N}}\{k\langle x\rangle{=}\underline{\underline{M}}\} \qquad \text{by lemma 3.3.2} \\
&= \underline{\underline{M[x \mapsto N]}}
\end{aligned}
$$

□

The above generalises to the recursive CPS calculus, so we have the following.

3.3.5 Lemma

$$\mathcal{N}(\!|M|\!)(m) = k\langle q\rangle\{q\langle\vec{z}\rangle{\Leftarrow}\mathcal{N}(\!|M|\!)(m)\{m\langle p\rangle{\Leftarrow}p\langle\vec{z}\rangle\}\}$$

3.3.6 Lemma

$$\mathcal{N}(\!|M[x \mapsto N]|\!)(m) = \mathcal{N}(\!|N|\!)(n)\{n\langle x\rangle{\Leftarrow}\mathcal{N}(\!|M|\!)(m)\}$$

3.3.7 Proposition

$$\mathcal{N}(\!|\lambda x.Mx|\!)(k) = \mathcal{N}(\!|M|\!)(k)$$

where $x \notin \mathsf{FV}(M)$.

3.3.8 Proposition

$$\mathcal{N}(\!|(\lambda x.M)N|\!)(k) = \mathcal{N}(\!|M[x \mapsto N]|\!)(k)$$

3.4 CPS transforms to the λ- and π-calculi

The CPS transforms with the CPS calculus as the target language can be specialised if we compose them with one of the translations of CPS calculus into other calculi.

For the λ-calculus, we recover the usual presentation of CPS transforms with the λ-calculus as the target language.

It has been noted by several people, such as [Bou97] and also [Thi96b], that translations from the λ- to the π-calculus (see [Tur95] for a survey of them, together with a CPS-like typing) can be seen as CPS transforms.

3.4.1 Remark With the the λ-calculus as the target language, the call-by-value CPS transform is the following:

$$
\begin{aligned}
(\!|x|\!) &= \lambda k.kx \\
(\!|\lambda x.M|\!) &= \lambda k.k(\lambda(x,h).(\!|M|\!)h) \\
(\!|\mathtt{throw}\ M\ N|\!) &= \lambda k.(\!|M|\!)(\!|N|\!) \\
(\!|\mathtt{callcc}\ M|\!) &= \lambda k.(\!|M|\!)(\lambda m.m(k,k)) \\
(\!|MN|\!) &= \lambda k.((\!|M|\!)(\lambda m.(\!|N|\!)(\lambda n.m(n,k))))
\end{aligned}
$$

3.4.2 Remark With the the λ-calculus as the target language, the Plotkin call-by-name CPS transform is the following:

$$\begin{aligned}
\mathcal{L}(\!|x|\!) &= \lambda k.xk \\
\mathcal{L}(\!|\lambda x.M|\!) &= \lambda k.k(\lambda(x,h).\mathcal{L}(\!|M|\!)h) \\
\mathcal{L}(\!|MN|\!) &= \lambda k.(\mathcal{L}(\!|M|\!)(\lambda m.m(\mathcal{L}(\!|N|\!),k)))
\end{aligned}$$

3.4.3 Remark With the the π-calculus as the target language, the call-by-value CPS transform is the following:

$$\begin{aligned}
(\!|x|\!)(k) &= \overline{k}\langle x\rangle \\
(\!|\lambda x.M|\!)(k) &= (\nu l)(\overline{k}\langle l\rangle \mid !\, l(xh).(\!|M|\!)(h)) \\
(\!|MN|\!)(k) &= (\nu m)((\!|M|\!)(m) \mid !\, m(l).(\nu n)((\!|N|\!)(n) \mid !\, n(a).\bar{l}\langle ak\rangle)) \\
(\!|\mathtt{throw}\ M\ N|\!)(k) &= (\nu m)((\!|M|\!)(m) \mid !\, m(n).(\!|N|\!)(n) \\
(\!|\mathtt{callcc}\ M|\!)(k) &= (\nu m)((\!|M|\!)(m) \mid !\, m(l).\bar{l}\langle kk\rangle)
\end{aligned}$$

where k, l, m, n are fresh.

3.4.4 Remark With the the π-calculus as the target language, the (lazy) call-by-name CPS transform is the following:

$$\begin{aligned}
\mathcal{L}(\!|x|\!)(k) &= \overline{x}\langle k\rangle \\
\mathcal{L}(\!|\lambda x.M|\!)(k) &= (\nu l)(\overline{k}\langle l\rangle \mid !\, l(xh).\mathcal{L}(\!|M|\!)(h) \\
\mathcal{L}(\!|MN|\!)(k) &= (\nu m)(\mathcal{L}(\!|M|\!)(m) \mid !\, m(l).(\nu n)(!\, n(h).\mathcal{L}(\!|N|\!)(h) \mid \bar{l}\langle nk\rangle))
\end{aligned}$$

3.4.1 Prompts and control-specific full abstraction

We would agree with [App92] that the intended meaning of (`throw`-ing to) a continuation is a "jump with arguments". But the CPS transform into the λ-calculus allows the definition of control operators that are rather unlike jumps, such as Felleisen's abort operator $\mathcal{A}$, definable as

$$(\!|\mathcal{A}M|\!) = \lambda k.((\!|M|\!)(\lambda x.x))$$

By ignoring the continuation k, $\mathcal{A}M$ discards the surrounding evaluation context and jumps to the top level. But this is not the same as a jump by `throw`-ing to some continuation previously reified by a `callcc`, as the destination of the jump changes when the phrase $\mathcal{A}M$ is enclosed in a bigger program.

Stranger still from the point of view that continuations are like jumps, Felleisen's prompt or control delimiter, definable as

$$(\!|\%M|\!) = \lambda k.k((\!|M|\!)(\lambda x.x))$$

can *intercept* jumps out of M.

We would argue that there is a fundamental difference between `callcc` and `throw` on the one hand and $\mathcal{A}$ and $\%$ on the other, certainly from a naive and probably from an implementation point of view. `callcc` is often described as labelling a control state, much as one can label a command in low-level imperative languages. `throw` is then analogous to `JMP` or `goto` in that it jumps to such a label. But whereas the label bindings introduced by `callcc` can be statically determined from the program text, $\mathcal{A}$ refers *dynamically* to the top level. The prompt $\%$, from this point of view, *destructively updates* all the references from within its argument to labels on the outside.

Whereas the definition of $\mathcal{A}$ for CPS into the λ-calculus looks innocuous, its very behaviour seems odd for π-calculus semantics. $\mathcal{A}$ is characterised by its discarding of the surrounding evaluation context $E[\ ]$, that is $(\!|E[\mathcal{A}M]|\!) = (\!|M|\!)$. But consider for instance $E[\ ] = (\lambda x.y)[\ ]$. Consider

$$(\!|(\lambda x.y)(\mathcal{A}M)|\!)(k) = (\nu n)(!\, l(xk).\overline{k}\langle y\rangle \mid (\!|\mathcal{A}M|\!)(n) \mid !\, n(a).\overline{l}\langle ak\rangle).$$

Here $(\!|\mathcal{A}M|\!)$ has no access at all to k. It is hard to see how this could ever be equal to $(\!|M|\!)(k)$. This would appear to be simply a violation of the visibility of names, reflecting the fact that the reference to the top level k that $\mathcal{A}M$ needs to escape is essentially dynamic, whereas π-calculus names are statically scoped.

Given that Sitaram and Felleisen [SF90] have shown that the prompt is necessary for the full-abstraction of standard CPS, it would be interesting to see whether the π-calculus semantics is a fully abstract translation. Continuation semantics in the π-calculus provides a different angle than the standard CPS model on the relative status of various control constructs. In the usual CPS semantics, direct style does not allow for control operators; and as soon as the interpretation is "bumped up" in the functional hierarchy by the CPS transform, the prompt is introduced along with `callcc` and `throw`. Felleisen and Sitaram [SF90] argue on the basis of this for the naturality of the prompt. In the π-calculus semantics, by contrast, the control operators already exist in the structure necessary to support the λ-calculus; and the prompt could be introduced only in a very imperative manner, by destructively updating the continuation of its subterm.

3.4.5 Remark We point out a connection between Milner's original encoding of the λ- in the π-calculus and a recently discovered CPS transform [HS97] for the $\lambda\mu$-calculus [Par92]. In each case, functions are not fully CPS transformed. Rather, a construct in the target language not properly inside the CPS discipline is used for the translation of functions. For the π-calculus, the parameter of the translation is used for *input*, whereas in CPS it would only ever be used for asynchronous output. Recall Milner's original transform [Mil91] from the λ- to the π-calculus (which he calls "encoding of the lazy λ-calculus"):

$$
\begin{array}{rcl}
\mathcal{L}(\!|x|\!)(k) & = & \overline{x}\langle k\rangle \\
\mathcal{L}(\!|\lambda x.M|\!)(k) & = & k(xh).\mathcal{L}(\!|M|\!)(h) \\
\mathcal{L}(\!|MN|\!)(k) & = & (\nu m)(\mathcal{L}(\!|M|\!)(m) \mid (\nu n)(\overline{m}\langle nk\rangle \mid !n(h).\mathcal{L}(\!|N|\!)(h)))
\end{array}
$$

The Hofmann-Streicher CPS transform $\widehat{(\_)}$ from the $\lambda\mu$-calculus is defined as follows.

$$
\begin{array}{rcl}
\widehat{x} & = & \lambda k.xk \\
\widehat{\lambda x.M} & = & \lambda(x,k).\widehat{M}k \\
\widehat{MN} & = & \lambda k.\widehat{M}(\widehat{N},k) \\
\widehat{\mu a.M} & = & \lambda a.\widehat{M}() \\
\widehat{[a]M} & = & \lambda().\widehat{M}a
\end{array}
$$

$$
\begin{array}{rcl}
\widehat{\sigma \to \tau} & = & \neg\widehat{\sigma} \times \widehat{\tau} \\
\widehat{\Gamma, x : \tau} & = & \widehat{\Gamma}, x : \neg\widehat{\tau}
\end{array}
$$

Here $\neg\tau$ abbreviates $\tau \to o$. The typing invariant of $\widehat{(\_)}$ is: if $\Gamma \vdash M : \tau$, then $\widehat{\Gamma} \vdash \widehat{M} : \neg\widehat{\tau}$. (CPS transforms are usually more comprehensible if one does *not* η-reduce them.) From our point of view, such transforms are somewhat vexing, in that they introduce continuations in some places, but do not appear to break down function types into continuations in quite the same way that the canonical CPS transforms do. It is not clear whether they can be accommodated in our semantic framework.

3.5 Flattening transforms

We review flattening transforms, known to be a first step towards CPS; see [LD93], although our account is somewhat different.

A first-order analogue of flattening for tuples leads us to a motivation for premonoidal categories: we derive them from the computationally natural principle of naming all intermediate results.

3.5.1 Flattening applications

We recall from [LD93] that CPS transforms can be staged by first translating into a flattened form (called Core Direct Style in [LD93]); here we use a `let`-language similar to that in [DH94].

3.5.1 Definition (Flattening transform for λ-terms)

$$\begin{array}{rcl} x^\flat & = & x \\ (\lambda x.M)^\flat & \stackrel{\text{def}}{=} & \lambda x.M^\flat \\ (M\,N)^\flat & \stackrel{\text{def}}{=} & \texttt{let}\ f = M^\flat\ \texttt{in}\ \texttt{let}\ a = N^\flat\ \texttt{in}\ f\,a \end{array}$$

3.5.2 Definition The CPS transforms for flat λ-terms are defined as follows.

$$\begin{array}{rcl} \overline{x} & = & k\langle x\rangle \\ \overline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\overline{M}\} \\ \overline{f a} & = & f\langle ak\rangle \\ \overline{\texttt{let}\ x = N\ \texttt{in}\ M} & = & \overline{N}\{k\langle x\rangle{=}\overline{M}\} \end{array}$$

$$\begin{array}{rcl} \underline{x} & = & x\langle k\rangle \\ \underline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\underline{M}\} \\ \underline{f a} & = & f\langle k\rangle\{k\langle f\rangle{=}f\langle ak\rangle\} \\ \underline{\texttt{let}\ x = N\ \texttt{in}\ M} & = & \underline{M}\{x\langle k\rangle{=}\underline{N}\} \end{array}$$

3.5.3 Proposition The call-by-value and the lazy call-by-name CPS transforms factor over flattening.

$$\overline{M^\flat} = \overline{M} \qquad \underline{M^\flat} = \underline{M}$$

3.5.2 Flattening tuples

In the above, the only compound expressions (or serious terms) were combinations $M\,N$. What the flattening transform achieved was to compile λ-calculus into a de-sugared form in which the only combined expressions were those made up of variables, $f\,a$.

We will now consider a first-order analogue of this, focussing on products instead of function spaces. We de-sugar (flatten) tuple expressions (M, N) in the same way as was done for applications $M\,N$. The target language shares some features with Moggi's metalanguage [Mog89], most notably in the distinction between values and computations, but without any reference to monads.

In this setting, we have a very restricted notion of value: tuples of variables. More complex entities cannot themselves be entries in a tuple; instead, all intermediate results are named and only the names can appear in tuples. Values are given by the following BNF:

$$V ::= () \mid x \mid (V, V)$$

Computations are values or `let`-expressions

$$M ::= V \mid \mathtt{let}\ x = M\ \mathtt{in}\ M$$

in general, though, M will range over other things in addition to the above, e.g. computations with side-effects.

3.5.4 Definition The typing of the flattened tuple language is given by the following rules.

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash U : \sigma \quad \Delta \vdash V : \tau}{\Gamma, \Delta \vdash (U, V) : \sigma \otimes \tau} \quad \text{where } U \text{ and } V \text{ are values}$$

$$\frac{\Gamma \vdash N : \sigma \quad \Delta, x : \sigma \vdash M : \tau}{\Delta, \Gamma \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : \tau} \qquad \frac{\Gamma \vdash N : \sigma \quad x : \sigma, \Delta \vdash M : \tau}{\Gamma, \Delta \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : \tau}$$

3.5.5 Definition The tuple flattening transform is defined as follows.

$$\begin{array}{rcl} x^\flat & \stackrel{\text{def}}{=} & x \\ (M, N)^\flat & \stackrel{\text{def}}{=} & \mathtt{let}\ a = M^\flat\ \mathtt{in}\ \mathtt{let}\ b = N^\flat\ \mathtt{in}\ (a, b) \end{array}$$

3.6 A duality on CPS terms

We recall the call-by-value and (lazy) call-by-name CPS transforms.

$$\begin{array}{rcl} \overline{x} & = & k\langle x\rangle \\ \overline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\overline{M}\} \\ \overline{MN} & = & \overline{M}\{k\langle f\rangle{=}\overline{N}\{k\langle x\rangle{=}f\langle xk\rangle\}\} \end{array}$$

$$\begin{array}{rcl} \underline{x} & = & x\langle k\rangle \\ \underline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\underline{M}\} \\ \underline{MN} & = & \underline{M}\{k\langle f\rangle{=}f\langle xk\rangle\{x\langle k\rangle{=}\underline{N}\}\} \end{array}$$

It is striking that the translations of a free variable x under the two transforms are dual to each other in the sense that each arises from the other by swapping operator and argument. The same duality is evident in the translations of the argument N in an application MN; this is $\overline{N}\{k\langle x\rangle{=}\ldots\}$ for call-by-value and $\ldots\{x\langle k\rangle{=}\underline{N}\}$ for call-by-name.

As long as continuations are unary, it is easy to define a transform that swaps operator and argument everywhere. This does not make sense for general, polyadic, continuations, as one cannot have a tuple in the operator position. But inasmuch as replacing $k\langle x\rangle$ with $x\langle k\rangle$ amounts to replacing x with a thunk, we can define a transformation for non-unary continuations $f\langle\vec{y}\rangle$ in the same spirit, by thunking them, giving $f\langle q\rangle\{q\langle p\rangle{=}p\langle\vec{y}\rangle\}$. To compensate for this thunking, the bindings for non-unary continuations $\ldots\{f\langle\vec{y}\rangle{=}\ldots\}$ need to be translated with an additional forcing $q\langle p\rangle$, giving $\ldots\{f\langle q\rangle{=}q\langle p\rangle\{p\langle\vec{y}\rangle{=}\ldots\}\}$

3.6.1 Definition For a CPS term M, let its *dual* $M^\perp$ be defined inductively as follows:

$$\begin{array}{rcl}
k\langle x\rangle^\perp & \stackrel{\text{def}}{=} & x\langle k\rangle \\
M\{n\langle x\rangle{=}N\}^\perp & \stackrel{\text{def}}{=} & N^\perp\{x\langle n\rangle{=}M^\perp\} \\
f\langle\vec{y}\rangle^\perp & \stackrel{\text{def}}{=} & f\langle q\rangle\{q\langle p\rangle{=}p\langle\vec{y}\rangle\} \\
M\{f\langle\vec{y}\rangle{=}N\}^\perp & \stackrel{\text{def}}{=} & M^\perp\{f\langle q\rangle{=}q\langle p\rangle\{p\langle\vec{y}\rangle{=}N^\perp\}\}
\end{array}$$

for $\vec{y} \neq x$; that is, $\vec{y}$ ranges over sequences other than those of unit length.

The duality between call-by-name and call-by-value is particularly vivid when we transform terms after they have been flattened.

3.6.2 Proposition For a CPS term M, $M^{\perp\perp} = M$

PROOF By induction on M. The cases $k\langle x\rangle$ and $M\{n\langle x\rangle{=}N\}$ are trivial. For the remaining two cases, we have

$$\begin{array}{rl}
& f\langle\vec{y}\rangle^{\perp\perp} \\
= & f\langle q\rangle\{q\langle p\rangle{=}p\langle\vec{y}\rangle\}^\perp \\
= & p\langle\vec{y}\rangle^\perp\{p\langle q\rangle{=}q\langle f\rangle\} \\
= & p\langle q\rangle\{q\langle f\rangle{=}f\langle\vec{y}\rangle\}\{p\langle q\rangle{=}q\langle f\rangle\} \\
= & f\langle\vec{y}\rangle
\end{array}$$

$$M\{f\langle\vec{y}\rangle{=}N\}^{\perp\perp}$$

$$
\begin{aligned}
&= M^{\perp}\{f\langle q\rangle{=}q\langle p\rangle\{p\langle\vec{y}\rangle{=}N^{\perp}\}\}^{\perp}\\
&= p\langle q\rangle\{p\langle q\rangle{=}q\langle f\rangle\{f\langle\vec{y}\rangle{=}N^{\perp\perp}\}\}\{q\langle f\rangle{=}M^{\perp\perp}\}\\
&= q\langle f\rangle\{f\langle\vec{y}\rangle{=}N^{\perp\perp}\}\{q\langle f\rangle{=}M^{\perp\perp}\}\\
&= M^{\perp\perp}\{f\langle\vec{y}\rangle{=}N^{\perp\perp}\}\\
&= M\{f\langle\vec{y}\rangle{=}N\} \qquad \text{by the induction hypothesis}
\end{aligned}
$$

□

This is a duality between call-by-value and call-by-name in the sense that it connects the corresponding CPS transforms.

3.6.3 Proposition For a λ-term M not containing control operators,

$$\overline{M}^{\perp} = \underline{M} \qquad \text{and} \qquad \underline{M}^{\perp} = \overline{M}$$

Proof By induction on M. For $M = x$,

$$\overline{x}^{\perp} = k\langle x\rangle^{\perp} = x\langle k\rangle = \underline{x}$$

And conversely. Let the induction hypothesis hold for M and N.

$$
\begin{aligned}
&\overline{M\,N}^{\perp}\\
&= \overline{N}\{k\langle x\rangle{=}\overline{M}\{k\langle f\rangle{=}f\langle xk\rangle\}\}^{\perp}\\
&= f\langle xk\rangle^{\perp}\{f\langle k\rangle{=}\overline{M}^{\perp}\}\{x\langle k\rangle{=}\overline{N}^{\perp}\}\\
&= f\langle q\rangle\{q\langle p\rangle{=}p\langle xk\rangle\}\{f\langle k\rangle{=}\overline{M}^{\perp}\}\{x\langle k\rangle{=}\overline{N}^{\perp}\}\\
&= \overline{M}^{\perp}\{k\langle p\rangle{=}p\langle xk\rangle\}\{x\langle k\rangle{=}\overline{N}^{\perp}\}\\
&= \underline{M}\{k\langle p\rangle{=}p\langle xk\rangle\}\{x\langle k\rangle{=}\underline{N}\}\\
&= \underline{M\,N}
\end{aligned}
$$

$$
\begin{aligned}
&\underline{M\,N}^{\perp}\\
&= \underline{M}\{k\langle f\rangle{=}f\langle xk\rangle\{x\langle k\rangle{=}\underline{N}\}\}^{\perp}\\
&= \underline{N}^{\perp}\{k\langle x\rangle{=}f\langle q\rangle\{q\langle p\rangle{=}p\langle xk\rangle\}\}\{f\langle k\rangle{=}\underline{M}^{\perp}\}\\
&= \underline{N}^{\perp}\{k\langle x\rangle{=}\underline{M}^{\perp}\{k\langle p\rangle{=}p\langle xk\rangle\}\}\\
&= \overline{N}\{k\langle x\rangle{=}\overline{M}\{k\langle p\rangle{=}p\langle xk\rangle\}\}\\
&= \overline{M\,N}
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\lambda x.M}^{\perp}\\
&= k\langle f\rangle\{f\langle xk\rangle{=}\overline{M}\}^{\perp}
\end{aligned}
$$

$$
\begin{aligned}
&= f\langle k\rangle\{f\langle q\rangle{=}q\langle p\rangle\{p\langle xk\rangle{=}\overline{M}^{\perp}\}\} \\
&= f\langle k\rangle\{f\langle q\rangle{=}q\langle p\rangle\{p\langle xk\rangle{=}\underline{M}\}\} \\
&= k\langle p\rangle\{p\langle xk\rangle{=}\underline{M}\} \\
&= \underline{\lambda x.M}
\end{aligned}
$$

$$
\begin{aligned}
&\underline{\lambda x.M}^{\perp} \\
&= k\langle f\rangle\{f\langle xk\rangle{=}\underline{M}\}^{\perp} \\
&= f\langle k\rangle\{f\langle q\rangle{=}q\langle p\rangle\{p\langle xk\rangle{=}\underline{M}^{\perp}\}\} \\
&= k\langle p\rangle\{p\langle xk\rangle{=}\underline{M}^{\perp}\} \\
&= k\langle p\rangle\{p\langle xk\rangle{=}\overline{M}\} \\
&= \overline{\lambda x.M}
\end{aligned}
$$

□

However, the duality does not in general respect equality, that is there are M_1 and M_2 with $M_1 = M_2$ (provable equality), but not ${M_1}^{\perp} = {M_2}^{\perp}$. Even for terms arising as CPS transform of λ-terms, equality is not preserved in general. Consider $(\lambda x.y)(fg)$.

$$\overline{(\lambda x.y)(fg)} = f\langle gx\rangle\{k\langle x\rangle{=}k\langle y\rangle\}$$

$$\underline{(\lambda x.y)(fg)} = y\langle k\rangle\{k\langle x\rangle{=}f\langle gx\rangle\} = k\langle y\rangle$$

Clearly, the Garbage Collection axiom is the culprit here, so perhaps restricting to the linear CPS calculus could help with the preservation of equality. Note that the duality is well-behaved with respect to the JMP and ETA laws, in that it transforms their redexes into each other.

$$(M[n \mapsto m])^{\perp} = (M\{n\langle x\rangle{=}m\langle x\rangle\})^{\perp} = x\langle m\rangle\{x\langle n\rangle{=}M^{\perp}\} = M^{\perp}[n \mapsto m]$$

3.7 Two connections between call-by-value and call-by-name

We have already mentioned in 3.1.5 that the lazy call-by-name CPS transform can be seen as arising from thunking.

Consider λ-calculus augmented by two special forms, `force` and `delay`. We give a translation from λ-calculus into the augmented variant.

$$
\begin{aligned}
x^{\mathsf{t}} &= \texttt{force}\,x \\
\lambda x.M^{\mathsf{t}} &= \lambda x.M^{\mathsf{t}} \\
(MN)^{\mathsf{t}} &= M^{\mathsf{t}}(\texttt{delay}\ N^{\mathsf{t}})
\end{aligned}
$$

We adapt the call-by-value CPS transform for special forms `force` and `delay` as follows.

$$\begin{aligned}\overline{\texttt{force}\,x} &= k\langle x\rangle \\ \overline{\texttt{delay}\ M} &= k\langle p\rangle\{p\langle k\rangle{=}\overline{M}\}\end{aligned}$$

Then call-by-name factors over call-by-value by virtue of thunking [HD95]: $\underline{M} = \overline{M^{\mathsf{t}}}$ (see [HD95]).

We note that the variables appearing in a source term are reused in its CPS transform, along with continuation variables freshly generated by the CPS transform. For different transforms, this reuse is conceptually quite different.

In call-by-value, the variables from the source language are recycled to denote the value of the translated term being passed to the current continuation. The latter is anonymous, inasmuch as the current one is always used. For instance, in $\overline{x} = k\langle x\rangle$, the variable x evaluates to itself; hence x is passed to the current continuation, which, in the shorthand version of the transform, is always called k. For $\overline{\lambda x.M} = k\langle f\rangle\{f\langle xk\rangle{=}\overline{M}\}$, the λ-expression evaluates to a closure; a pointer (private name) to this is passed to the current continuation.

For call-by-name, there are two intuitively different, but equivalent readings. Either one of these may seem more natural, depending on whether one looks at the thunking or the flattening transform as an intermediate step towards call-by-name CPS. They are equivalent in that they are adjoint correspondent under the self-adjointness on the left.

In the first interpretation, which regards call-by-name as a variant of call-by-value obtained by thunking, λ-calculus variables are recycled in the manner described above for call-by-value. The difference that the environment does not hold values, but thunks. Hence $\underline{x} = k\langle x\rangle$ is read as saying that the thunk whose address x is stored in the environment is forced by being sent the current continuation k. (Categorically, then, $[\![x:\tau\vdash x:\tau]\!]$ denotes a morphism $\neg\neg[\![\tau]\!] \longrightarrow [\![\tau]\!]$ representing this forcing.) According to this view, the translation of λ-expressions is identical to that under call-by-value.

The second interpretation of call-by-name holds that source language variables are recycled to denote the current continuation of the translated expression. The latter, rather than the current continuation, now becomes an anonymous request channel or return address. $\underline{\underline{x}} = k\langle x\rangle$ is read as saying that k is passed to the current continuation, called x, of the transform of the λ-term x. (Categorically, this means that $[\![x:\tau\vdash x:\tau]\!]$ denotes a morphism $\neg[\![\tau]\!] \longrightarrow \neg[\![\tau]\!]$ representing the identity.) Under this interpretation, application in particular is quite different from the call-by-value case.

$$\underline{fx} = f\langle q\rangle\{q\langle p\rangle{=}p\langle xk\rangle\}$$

f is now the current continuation of application, to which the request channel q is passed.

3.8 From flattening tuples to premonoidal categories

One could argue that the tuple notation (M, N) in a call-by-value language should be considered as no more than syntactic sugar for the flattened form

$$\texttt{let}\ a = M\ \texttt{in}\ \texttt{let}\ b = N\ \texttt{in}\ (a, b)$$

and that semantics should be based on the de-sugared form. Thus semantics should not be based on the categorical structure for which the tuple language $M ::= () \mid x \mid (M, M)$ is the internal language, finite products [Cro93]; but instead on the structure corresponding to the de-sugared `let`-language.

In the spirit of categorical semantics (for an accessible introduction, see e.g. the textbook [Cro93]), we now attempt to arrive at a categorical semantics.

The minimal setting for a semantics of the flattened tuple language is a category equipped with a "tensor", more precisely, a binoidal category [PR97].

The `let`-construct is decomposed into tensor $A \otimes (\_)$ and composition. That is, in a judgement

$$\Delta, \Gamma \vdash \texttt{let}\ x = N\ \texttt{in}\ M : \tau$$

$[\![\Delta]\!]$ is composed "in parallel" with $[\![\Gamma \vdash N : \sigma]\!]$ by means of the tensor $[\![\Delta]\!] \otimes (\_)$, and the result is composed "sequentially" with $[\![\Delta, x : \sigma \vdash M : \tau]\!]$ by means of the categorical composition ";".

In order to make the semantics cope with ambiguities in the notation, specifically writing environments as associative lists, we require coherence conditions that make the different readings of ambiguous syntax agree semantically. This leads to the notion of premonoidal category [PR97].

3.8.1 Definition Given a premonoidal category and an interpretation $[\![\_]\!]$ of base types as objects of that category, we give a semantics to the flattened tuples language as follows.

$$[\![x : \tau_1, \ldots x_n : \tau_n]\!] \quad \stackrel{\text{def}}{=} \quad (\ldots([\![\tau_1]\!] \otimes \ldots)\ldots) \otimes [\![\tau_n]\!]$$

$$
\begin{aligned}
[\![x:\tau \vdash x:\tau]\!] &\stackrel{\text{def}}{=} \mathsf{id} \\
[\![\Gamma,\Delta \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : \tau]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash N:\sigma]\!] \otimes [\![\Delta]\!]; [\![x:\sigma,\Delta \vdash M:\tau]\!] \\
[\![\Delta,\Gamma \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : \tau]\!] &\stackrel{\text{def}}{=} [\![\Delta]\!] \otimes [\![\Gamma \vdash N:\sigma]\!]; [\![\Delta, x:\sigma \vdash M:\tau]\!] \\
[\![\Gamma,\Delta \vdash (U,V) : \sigma\otimes\tau]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash U:\Gamma]\!] \otimes [\![\Delta]\!]; [\![\sigma]\!] \otimes [\![\Delta \vdash V:\tau]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![x:\tau_1,(y:\tau_2,z:\tau_3) \vdash ((x,y),z) : (\tau_1\otimes\tau_2)\otimes\tau_3]\!] &\stackrel{\text{def}}{=} \mathsf{assoc} \\
[\![(x:\tau_1,y:\tau_2),z:\tau_3 \vdash (x,(y,z)) : \tau_1\otimes(\tau_2\otimes\tau_3)]\!] &\stackrel{\text{def}}{=} \mathsf{assoc}^{-1}
\end{aligned}
$$

The coherence theorems [PR97] for premonoidal categories then let us write environments associatively. The ambiguities in the syntax do not lead to problems because "every diagram commutes" (as long as it is made up of the denotations of values).

The `let` expression can be written almost exactly the same in ML (`let val` x `=` N `in` M `end`). By way of illustration, consider the following ML code.

```
fun assoc ((x,y),z) = (x,(y,z));
    assoc : ('a * 'b) * 'c -> 'a * ('b * 'c);

fun tensorleft f (a,x) = (a, f x);
    tensorleft : ('a -> 'b) -> 'c * 'a -> 'c * 'b;

fun tensorright f (x,a) = (f x, a);
    tensorright : ('a -> 'b) -> 'a * 'c -> 'b * 'c;
```

In Scheme, we would write flattened tuple expressions using `let*` like this: `(let* ((`a M`) (`b N`)) (list (`a b`)))`. Note that the heterogeneous lists in Scheme allow us to define a *strict* premonoidal category by using list concatenation for defining $\otimes$.

Some readers may be surprised that the premonoidal structure is not in fact monoidal. We briefly illustrate why one should not expect this. For example, in a language with state, there are two possible meanings of a tuple (M,N), depending which component is evaluated first. Consider the following examples, where we make the evaluation order explict by using `let`.

```
let val s = ref 0 in
let val x = (s := !s + 1; !s) in
```

```
let val y = (s := !s + 1; !s)
in #1(x,y) end end end ;

let val s = ref 0 in
let val y = (s := !s + 1; !s) in
let val x = (s := !s + 1; !s)
in #1(x,y) end end end;
```

Just as for state, in the presence of continuations (first-class or otherwise) there are two possible meanings of the tuple `(throw k 1, throw k 2)`.

```
callcc(fn k =>
       let val x = throw k 1 in
       let val y = throw k 2
       in #1(x,y) end end);

callcc(fn k =>
       let val y = throw k 2 in
       let val x = throw k 1
       in #1(x,y) end end);
```

In a monoidal category, there would be no way to distinguish between the two composites. This makes monoidal categories suitable for those cases where both composites are evaluated in parallel *or* where there can be no interference between the two (which would the case, say, if both had access to disjoint pieces of state). But with control, as given by continuations, we have both a sequential evaluation order and interference between the components, since a jump in one will prevent the other from being evaluated at all.

Put differently, the presence of computational effects, like state and control, "breaks" the bifunctoriality, so one is left with a binoidal category. (Partiality appears to be a separate case that should perhaps not be lumped together with genuine effects like state and control.)

Chapter 4

$\otimes\neg$-categories

In this chapter, we develop a categorical account of the structure inherent in first-class continuations.

For first-class continuations, it is particularly worthwhile to look at the category of computations, for the following reasons:

- First-class continuations allow the full `callcc` to be added to the language, which is the most powerful version of control found in actual languages. This contrasts with the situation for state, where only a rather weak notion of global state is added by commonly used notions like the state monad.
- The construct to be studied has universal properties on the category of computations. That does not seem to be the case for constructs associated with state, such as assignment.
- Continuations are an advanced concept in programming languages that could be made easier to use by semantic clarification. While local state has subtleties, it is not obvious if *global* variables as introduced by the state monad are all that much in need of elucidation.

(We have made a comparsion with state here, as state and control appear to be the most natural things to add to a programming language, but this discussion would apply to other effects, e.g. exceptions.)

4.1 Introduction: what structure do we need?

The task of finding a semantic infrastructure for continuation semantics is somewhat analogous to that of interpreting λ-calculus in a cartesian closed category. We need a first-order structure for interpreting environments and tuple types, in analogy with, but weaker than, cartesian products, as well as higher-order structure for interpreting continuation types. These now become the fundamental notion, while arrow types are derived as a special instance of continuation types. But whereas in λ-calculus every morphism is a "pure function", in CPS there is a need to identify a subcategory of effect-free computations (or values) that satisfy stronger properties than the general, possibly effectful, computations.

We show that effect-freeness in the presence of *first-class* continuations is a more subtle notion than would at first appear. In particular, it is not enough to exclude straightforward jumps like `throw k 42`.

In our framework, environments are modelled by means of a premonoidal category [PR97]: this is a categorical framework which provides enough parallelism on types to accommodate programs of multiple arity, but no real parallelism on programs. For each object (type) A, there are functors $A \otimes (\_)$ and $(\_) \otimes A$. For morphisms $f : A \longrightarrow A'$ and $g : B \longrightarrow B'$, there are in general two *distinct* parallel compositions, $f \otimes B; A' \otimes g$ and $A \otimes g; f \otimes B'$. The *central* morphisms are those f such that for all g, the above composites agree. That is, those programs phrases which are not sensitive as to whether they are evaluated before or after any other. This provides a robust notion of effect-free morphism.

The continuation type constructor extends to a contravariant functor, as every function $\sigma \rightarrow \tau$ gives rise to a continuation transformer $\tau\,\mathtt{cont} \rightarrow \sigma\,\mathtt{cont}$ in the opposite direction. This functor is adjoint to itself on the left, i.e., there is a natural bijection

$$\frac{\neg B \longrightarrow C}{\neg C \longrightarrow B}$$

We use the notation $\neg$ and refer to an application of the continuation functor more succintly as a negation, without claiming any deep connection. [1]

Intuitively, a morphism $\neg B \longrightarrow C$ expects both a B- and a C-accepting continuation as its argument and current continuation, respectively. The above correspondence arises by simply switching these. This ties in with the typing of continuations in Standard ML of New Jersey.

We can define the unit $\mathsf{force} : \neg\neg A \longrightarrow A$ of this adjunction, the isomorphism of adjunction $\phi : \mathrm{hom}(\neg B, C) \longrightarrow \mathrm{hom}(\neg C, B)$ and the negation functor itself.

We require this to hold even "parametrically" in some other object A

$$\frac{A \otimes \neg B \longrightarrow C}{A \otimes \neg C \longrightarrow B}$$

[1] There is a formal resemblance between the continuation functor and logical negation, just as there is a formal resemblance between, say, slice categories and division on the integers by virtue of $\mathcal{C}/\mathbf{1} \cong \mathcal{C}$ and $\mathcal{C}/A/B \cong \mathcal{C}/(A \times B)$.

The unit of this adjunction is the application map $\mathsf{apply} : A \otimes \neg(A \otimes \neg B) \longrightarrow B$. Restricted to the subcategory of central morphisms, $\neg$ is adjoint to itself on the right.

$$\frac{A \longrightarrow \neg B}{B \longrightarrow \neg A}$$

Intuitively, a *central* morphism $A \longrightarrow \neg B$ expects an argument of type A and returns a B-accepting continuation. Hence there is demand for both A and B; and again the correspondence arises essentially by swapping.

The unit of this adjunction is a generic delaying map $\mathsf{thunk} : A \longrightarrow \neg\neg A$.

Using thunk, we define a morphism

$$\mathsf{pair} : C \longrightarrow \neg(A \otimes \neg(A \otimes C))$$

which is a natural transformation in the centre. This in turn is used to define λ-abstraction.

$$\overline{\lambda}_A f \stackrel{\text{def}}{=} \mathsf{pair}; \neg(A \otimes \neg f)$$

This definition of λ-abstraction in terms of control (and tuple types) does not give rise to a closed category, although we have the following.

$$(A \otimes \overline{\lambda} f); \mathsf{apply} = A \otimes \mathsf{pair}; A \otimes \neg(A \otimes \neg f); \mathsf{apply} = A \otimes \mathsf{pair}; \mathsf{apply}; f = f$$

The corresponding $\overline{\lambda}(A \otimes g; \mathsf{apply}) = g$, however does not hold in general. Hence this λ-abstraction does not give rise to a cartesian closed category. But it is still sufficient for interpreting a call-by-value λ-calculus, as a central g can be pushed into λ (and values denote central morphisms). Although neither SML nor Scheme make this identification of function types $[A \to B]$ with $\neg(A \otimes \neg B)$, we can still define a pair of coercion functions (figures 1.12 and 1.13).

The basis for our categorical account of continuation semantics will be the negation functor, corresponding to the typing based on $\neg$ in Chapter 2 However, the continuations considered there were actually *polyadic*, that is, in $k\langle x_1 \dots x_i \rangle$ k is applied to a tuple of arguments. That is why, before introducing $\neg$, we need some first-order structure for building up such tuples (as well as environments).

4.2 Semantics of environments in a premonoidal category

Before addressing the categorical semantics of environments, we should perhaps clarify what we mean by "environment" here. The terminology we adopt may not be completely standard, but is a rational one in being semantically motivated.

In Type Theory, the antecedent of a judgement $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau$ is usually called a context. In Computer Science, an environment is traditionally a map from variables to values. Here we use the word environment in the general sense of anything that ascribes something (types, values, ...) to variables; the former, then, is a *type environment* and the latter a *value environment*.

Semantically, a type environment denotes an object in some semantic category, while a value environment denotes an element thereof. (In the categorical sense of element: an element of an object being a morphism with that object as its codomain.)

This generalises the usual notion of types denoting objects and terms denoting their elements in a straightforward pointwise fashion to indexed collections of both: value and type environments, respectively.

In the sequel, we concentrate on the semantics of type environments and do not deal with value environments explicitly. But their semantics is implicit in that a morphism $[\![\Gamma]\!] \longrightarrow [\![\tau]\!]$ can be seen as mapping elements of $[\![\Gamma]\!]$ to elements of $[\![\tau]\!]$.

In particular, suppose our semantic category consists of sets with structure, and that we build up the denotation of a type environment $[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]$ as the product of the denotations of the types $[\![\tau_i]\!]$. Then a value environment ρ is (up to isomorphism) a (global) element of the denotation of the type environment

$$\mathbf{1} \xrightarrow{\rho} [\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

This also implies a notion of value environments "matching" type environments in that ρ ascribes to each x_i a value v_i having type τ_i.

The above generalisation of environments is also closely related to the view of structures in Standard ML having the signatures which they match as their "type" [FT95], to the extent that an ML structure represents a value environment and a signature a type environment.

Another reason for avoiding the word "context" here is that in Computer Science this often refers to a notion of "term with a hole", such as evaluation context. Using "context" in the sense of "type environment", moreover, may lead to an unfortunate inversion of terminology if the word "environment" is then used for (the totality of) computing agents with which some interaction is possible, a concept more closely allied with "evaluation context" or indeed "continuation".

We will use a premonoidal structure $\otimes$ for interpreting environments of the form $[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]$ as $[\![\tau_1]\!] \otimes \cdots \otimes [\![\tau_n]\!]$.

4.2.1 Definition ([PR97]) A strict premonoidal category is a category $\mathcal{K}$ together with an object $\mathbf{1} \in \mathbf{Ob}\mathcal{K}$ and, for each $A \in \mathbf{Ob}\mathcal{K}$, endofunctors $A \otimes (\_)$ and $(\_) \otimes A$ that agree in the sense that

$$(A \otimes (\_))(B) = ((\_) \otimes B))(A) =: A \otimes B$$

such that $\mathbf{1} \otimes (-) = \mathsf{id}_{\mathcal{K}} = (-) \otimes \mathbf{1}$ and

$$\begin{aligned}
(A \otimes B) \otimes C &= A \otimes (B \otimes C) \\
(f \otimes B) \otimes C &= f \otimes (B \otimes C) \\
(A \otimes g) \otimes C &= A \otimes (g \otimes C) \\
(A \otimes B) \otimes h &= A \otimes (B \otimes h)
\end{aligned}$$

A morphism $f : A \longrightarrow A'$ is called central if it commutes with everything in the sense that, for all $g : B \longrightarrow B'$, we have

$$\begin{aligned}
A \otimes g; f \otimes B' &= f \otimes B; A' \otimes g \\
g \otimes A; B' \otimes f &= B \otimes f; g \otimes A'
\end{aligned}$$

That is, for f to be central, we require these diagrams to commute

$$\begin{tikzcd}
A \otimes B \arrow[r, "A\otimes g"] \arrow[d, "f\otimes B"'] & A \otimes B' \arrow[d, "f\otimes B'"] \\
A' \otimes B \arrow[r, "A'\otimes g"] & A' \otimes B'
\end{tikzcd}
\qquad
\begin{tikzcd}
B \otimes A \arrow[r, "g\otimes A"] \arrow[d, "B\otimes f"'] & B' \otimes A \arrow[d, "B'\otimes f"] \\
B \otimes A' \arrow[r, "g\otimes A'"] & B' \otimes A'
\end{tikzcd}$$

The centre $\mathbf{Z}(\mathcal{K})$ of $\mathcal{K}$ is the subcategory of $\mathcal{K}$ consisting of all objects and all central morphisms. Let $\iota : \mathbf{Z}(\mathcal{K}) \hookrightarrow \mathcal{K}$ be the inclusion.

The inclusion of the centre will often be left implicit.

4.2.2 Remark To simplify the account, we have concentrated on the strict case, rather than the more general premonoidal category. Because each premonoidal category is equivalent to a *strict* premonoidal category (implicit in [PR97]), the restriction to strictness is not a very severe one.

One could reasonably expect everything to generalise to the general case in a routine way.

In any case, the emphasis here is on the categorical structure of continuations, so we believe it to be defensible to postpone coherence issues until this is well-understood and definitive. — For the present, the canonical example is a term model (which we try to understand more abstractly), so it would be somewhat counterproductive not to take advantage of the strictness afforded by term models.

4.2.3 Definition A $\otimes$-category is a strict premonoidal category $\mathcal{K}$ such that $\otimes$ is given by cartesian product in the centre of $\mathcal{K}$ and furthermore, the twist map arising from this product

$$\langle \pi_2, \pi_1 \rangle : A \otimes B \longrightarrow B \otimes A$$

is natural in A and and in B.

We extend the morphism pairing operation $\langle \_, \_ \rangle$ given by the products in the centre of $\mathcal{K}$ to the whole of $\mathcal{K}$ as follows. (Note that this implies a choice of which component is computed first: here it is the *second*.) For $f : C \longrightarrow A$ and $g : C \longrightarrow B$, let

$$\langle f, g \rangle \stackrel{\text{def}}{=} \langle \mathsf{id}_C, \mathsf{id}_C \rangle; C \otimes g; f \otimes B \quad : \quad C \longrightarrow A \otimes B$$

4.2.4 Definition We say that a morphism $f : A \longrightarrow B$ is copyable iff it respects the binary products of the centre in the sense that

$$f; \langle \mathsf{id}, \mathsf{id} \rangle = \langle f, f \rangle : A \longrightarrow B \otimes B$$

and that f is discardable if it respects the terminal object $\mathbf{1}$ of the centre in the sense that

$$f; !_B = !_A : A \longrightarrow \mathbf{1}$$

In [Fil89], discardable morphisms are called total.

4.3 Continuation types as higher-order structure

We will be interested in a particularly simple kind of adjunction: a contravariant functor being adjoint to its own dual, with the unit and co-unit being the same.

4.3.1 Definition A functor $F : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}$ is called *self-adjoint on the left* iff there is a natural transformation $\epsilon : FF^{\mathrm{op}} \longrightarrow \mathsf{id}_{\mathcal{C}}$ such that $F\epsilon; \epsilon F = \mathsf{id}$. Dually, F is called *self-adjoint on the right* iff F^{op} is self-adjoint on the left.

(See also [Mac71, p. 87] for the "on the left" idiom.)

The continuation functor $\neg$ has two universal properties, adjointness on the left and right; we axiomatise them here in terms of the universal maps apply and thunk, respectively.

4.3.2 Definition A $\otimes\neg$-category is a $\otimes$-category $\mathcal{K}$ together with

- a functor $\neg : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathsf{Z}(\mathcal{K})$ such that for each object A of $\mathcal{K}$,

$$(A \otimes \iota\neg(\_)) : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathcal{K}$$

  is self-adjoint on the left (let $\mathsf{apply}_A : A \otimes \neg(A \otimes \neg B) \longrightarrow B$ be the unit of this adjunction), and

- a natural transformation $\mathsf{thunk} : \mathsf{id}_{\mathsf{Z}(\mathcal{K})} \longrightarrow \neg\neg$ in $\mathsf{Z}(\mathcal{K})$

such that

- apply is dinatural in A
- $\mathsf{thunk}; \mathsf{force} = \mathsf{id}$ where $\mathsf{force} \stackrel{\mathrm{def}}{=} \mathsf{apply}_{\mathbf{1}} : \neg\neg A \longrightarrow A$
- $\neg\,\mathsf{force} = \mathsf{thunk}\,\neg$
- letting $\underline{\underline{\mathsf{apply}}}_A \stackrel{\mathrm{def}}{=} A \otimes \neg(A \otimes \mathsf{force}); \mathsf{apply}_A$, we have

$$\begin{aligned}
\neg\,\mathsf{force} &= \mathsf{thunk}\,\neg \\
\mathsf{thunk}; \neg\neg\underline{\underline{\mathsf{apply}}} &= \underline{\underline{\mathsf{apply}}}; \mathsf{thunk} \\
\mathsf{thunk}_{A\otimes C} &= A \otimes \mathsf{thunk}_C; A \otimes \neg\underline{\underline{\mathsf{apply}}}; \underline{\underline{\mathsf{apply}}} \\
\mathsf{apply}_{A\otimes A'} &= \langle \pi_2, \pi_1 \rangle \otimes \neg(A \otimes A' \otimes \neg B); A' \otimes \underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_{A'}
\end{aligned}$$

The first of these four axioms establishes another link between forcing and thunking (in addition to the more familiar $\mathsf{thunk}; \mathsf{force} = \mathsf{id}$); the second states that the call-by-name application, unlike the call-by-value one, is effect-free; the other two are somewhat technical coherence conditions.

Intuitively, dinaturality of the application map means that modifying the operand of a function application by a map $f : A \longrightarrow A'$ is the same as modifying the operator by a corresponding continuation transformer.

$$\begin{tikzcd}
A \otimes \neg(A' \otimes \neg B) \arrow[r, "A\otimes\neg(f\otimes\neg B)"] \arrow[d, "f\otimes\neg(A'\otimes\neg B)"'] & A \otimes \neg(A \otimes \neg B) \arrow[d, "\mathsf{apply}"] \\
A' \otimes \neg(A' \otimes \neg B) \arrow[r, "\mathsf{apply}"] & B
\end{tikzcd}$$

Dinaturality is rquired as a separate axiom, as it does not seem to follow from naturality (unlike in a Cartesian closed category).

The universal property of the continuation functor can be expressed by the following diagrams (naturality and triangular identity for force.)

$$\begin{tikzcd}
\neg\neg A \arrow[r, "\mathsf{force}"] \arrow[d, "\neg\neg f"'] & A \arrow[d, "f"] \\
\neg\neg B \arrow[r, "\mathsf{force}"] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
\neg A \arrow[r, "\neg\mathsf{force}"] \arrow[dr, "\mathsf{id}"'] & \neg\neg\neg A \arrow[d, "\mathsf{force}"] \\
 & \neg A
\end{tikzcd}$$

In addition to the usual $\mathsf{thunk}; \mathsf{force} = \mathsf{id}$, we have another axiom linking forcing and thunking. A consequence of this is the self-adjointness on the right of the restriction of $\neg$ to the centre, with unit thunk.

$$
\begin{tikzcd}
A \arrow[r, "\mathsf{thunk}"] \arrow[d, "g"'] & \neg\neg A \arrow[d, "\neg\neg g"] \\
B \arrow[r, "\mathsf{thunk}"] & \neg\neg B
\end{tikzcd}
\qquad
\begin{tikzcd}
\neg A \arrow[r, "\mathsf{thunk}"] \arrow[dr, "\mathsf{id}"'] & \neg\neg\neg A \arrow[d, "\neg\mathsf{thunk}"] \\
& \neg A
\end{tikzcd}
$$

(where g is central.) In chapter 6 (figures 6.3 and 6.4), we will consider programs written in "compositional" style, that is, using sequential composition of functions and the programming analogues of thunk, $\neg$, ... (figures 1.12 and 1.13). We hope that the simple and quite symmetric categorical laws expressed by the above diagrams could facilitate reasoning about programs written in this style.

4.3.3 Definition Given a cartesian closed category $\mathcal{C}$ (with strict products), and an object R of $\mathcal{C}$ we can define a $\otimes\neg$-category $\mathcal{K}$ as follows

$$
\begin{aligned}
\mathbf{Ob}\,\mathcal{K} &\stackrel{\text{def}}{=} \mathbf{Ob}\,\mathcal{C} \\
\mathcal{K}(A, B) &\stackrel{\text{def}}{=} \mathcal{C}(R^B, R^A)
\end{aligned}
$$

$A \otimes (\_)$ is given by the product $A \times (\_)$ in $\mathcal{C}$. The functor $\neg$ is $R^{(\_)}$. $\mathsf{force} \stackrel{\text{def}}{=} \eta R^{(\_)}$ and $\mathsf{thunk} \stackrel{\text{def}}{=} R^{\eta}\eta R^{(\_)}$, where $\eta_A : A \longrightarrow R^{R^A}$ is the unit of the "continuation monad" on $\mathcal{C}$. We call $\mathcal{K}$ the standard model for $\mathcal{C}$ and R.

Despite the apparent generality of this construction, we regard this as an overly specific approach that does not do justice to the full generality of CPS (compare section 2.6). It consists essentially of implementing CPS in simply-typed λ-calculus and then interpreting this in the usual fashion in a cartesian closed category.

4.3.4 Remark In the category $\mathcal{C}$, the functor $[(\_) \to R]$ is self-adjoint on the right. The two isomorphic views of the continuation semantics category as a category of continuation transformers and as the Kleisli category of the continuation monad are connected by the self-adjointness (i.e. its isomorphism of adjunction).

$$\mathcal{C}(A, R^B) \cong \mathcal{C}(B, R^A)$$

This is also the connection between the typings of the Plotkin-style (continuation last) and Fischer-style (continuation first) CPS transforms. On types, this gives the continuation monad view $\overline{\sigma \to \tau} = \overline{\sigma} \to \neg\neg\tau$, or the continuation transformer view $\overline{\sigma \to \tau} = \neg\overline{\tau} \to \neg\sigma$, respectively.

4.4 Some interdependencies of properties

We write ϕ_A for the isomorphism of adjunction for the self-adjointness on the left. This is an involution. $\phi_A : \mathcal{K}(A \otimes \neg B, C) \longrightarrow \mathcal{K}(A \otimes \neg C, B)$

$\phi_A f \stackrel{\text{def}}{=} A \otimes \neg f; \mathsf{apply}_A$

For $A = \mathbf{1}$, we have $\phi \stackrel{\text{def}}{=} \phi_{\mathbf{1}} : \mathcal{K}(\neg B, C) \longrightarrow \mathcal{K}(\neg C, B)$ with $\phi f = \neg f; \mathsf{force}$.

4.4.1 Remark Note that because of the finite product structure on the centre, each functor $A \otimes (\_)$ comes equipped with a comonad structure, the unit being given by discarding, the multiplication by weakening.

We can regard ϕ_A as essentially the same as ϕ, but on the co-Kleisli category for the comonad $A \otimes (\_)$

$$\begin{aligned} \mathbf{Ob}\,\mathcal{K}_A &\stackrel{\text{def}}{=} \mathbf{Ob}\,\mathcal{K} \\ \mathcal{K}_A(B, C) &\stackrel{\text{def}}{=} \mathcal{K}(A \otimes B, C) \end{aligned}$$

For each A, $\mathcal{K}$ has its own "indexed negation" $\neg_A$, defined as

$$\neg_A f \stackrel{\text{def}}{=} \phi_A(A \otimes \mathsf{force}; f).$$

This "indexed functor" point of view has some advantages. In particular, some of the axioms of an $\otimes\neg$-category become more comprehensible: they were essentially reverse-engineered to make $\neg_A$ viable as an indexed functor. This issue will be addressed more fully in chapter 8 where we take the indexed category presentation as fundamental.

4.4.2 Remark What is perhaps surprising about this definition is that we have made such strong assumptions about the centre. All central morphisms are deemed to be effect-free, so that they respect the product. While centrality is certainly necessary for effect-freeness, there is in general no reason to assume that it is sufficient. It appears to be the presence of first-class continuations, specifically the unit force, that that makes centrality such a strong property: if a morphism commutes with everything, it must commute with force, and that implies that it commutes with reification. Slightly more technically, if $f : A \longrightarrow B$ is central, then $f \otimes \neg\neg\neg B; B \otimes \mathsf{force} = A \otimes \mathsf{force}; f \otimes \neg B$. This implies the naturality of thunk, as

$$\begin{aligned} & f; \mathsf{thunk} \\ = \;& A \otimes \mathsf{thunk}; A \otimes \neg(f \otimes \neg\neg\neg B; B \otimes \mathsf{force}; \underline{\underline{\mathsf{apply}}}); \mathsf{apply} \\ = \;& A \otimes \mathsf{thunk}; A \otimes \neg(A \otimes \mathsf{force}; f \otimes \neg B; \underline{\underline{\mathsf{apply}}}); \mathsf{apply} \\ = \;& \mathsf{thunk}; \neg\neg f \end{aligned}$$

4.4.3 Remark If a morphism f is thunkable in the sense that it makes the naturality square

$$\begin{tikzcd} A \arrow[r, "\mathsf{thunk}"] \arrow[d, "f"'] & \neg\neg A \arrow[d, "\neg\neg f"] \\ B \arrow[r, "\mathsf{thunk}"] & \neg\neg B \end{tikzcd}$$

commute, then it respects binary products.

$$\begin{array}{rl} & f;\langle\mathsf{id},\mathsf{id}\rangle \\ = & f;\langle\mathsf{thunk};\mathsf{force},\mathsf{thunk};\mathsf{force}\rangle \\ = & f;\mathsf{thunk};\langle\mathsf{force},\mathsf{force}\rangle \\ = & \mathsf{thunk};\neg\neg f;\langle\mathsf{force},\mathsf{force}\rangle \\ = & \langle\mathsf{thunk};\neg\neg f;\mathsf{force},\mathsf{thunk};\neg\neg f;\mathsf{force}\rangle \\ = & \langle f,f\rangle \end{array}$$

4.4.4 Remark Instead of defining $\neg$ to have the centre as it codomain, we could have required the adjoint correspondence

$$\frac{A\otimes\neg B\longrightarrow C}{A\otimes\neg C\longrightarrow B}$$

to be natural in A, as this implies that every morphism of the form $\neg f$ is central. This property is perhaps more intuitive in terms of control flow: control manipulation concerning B and C does not affect a separate strand of control $g : A \longrightarrow A'$.

4.4.5 Proposition Every negated morphism $\neg C \xrightarrow{\neg f} \neg B$ is central.

Proof Let $B \xrightarrow{f} C$ and $A \xrightarrow{g} A'$. We need to show that

$$A\otimes\neg C \xrightarrow{A\otimes\neg f} A\otimes\neg B \xrightarrow{g\otimes\neg B} A'\otimes\neg B$$

and

$$A\otimes\neg C \xrightarrow{g\otimes\neg C} A'\otimes\neg C \xrightarrow{A'\otimes\neg f} A'\otimes\neg B$$

are the same morphism. Applying ϕ_A to the first composite, we get

$$\begin{array}{rl} & \phi_A(A\otimes\neg f;g\otimes\neg B) \\ = & \phi_A(A\otimes\neg f;g\otimes\neg B;\mathsf{id}_{A'\otimes\neg B}) \\ = & \phi_A(g\otimes\neg B;\mathsf{id}_{A'\otimes\neg B});f \\ = & g\otimes\neg(A'\otimes\neg B);\phi_{A'}(\mathsf{id}_{A'\otimes\neg B});f \end{array}$$

For the second composite:

$$\begin{array}{rl} & \phi_A(g \otimes \neg C; A' \otimes \neg f) \\ = & \phi_A(g \otimes \neg C; A' \otimes \neg f; \mathsf{id}_{A' \otimes \neg B}) \\ = & g \otimes \neg(A' \otimes \neg B); \phi_{A'}(A' \otimes \neg f; \mathsf{id}_{A' \otimes \neg B}) \\ = & g \otimes \neg(A' \otimes \neg B); \phi_{A'}(\mathsf{id}_{A' \otimes \neg B}); f \end{array}$$

Because ϕ_A is an isomorphism, this means that $A \otimes \neg f; g \otimes \neg B = g \otimes \neg C; A' \otimes \neg f$.

□

4.4.6 Proposition

$$\begin{tikzcd} A' \otimes A \otimes \neg(A \otimes A' \otimes \neg B) \arrow[r, "A' \otimes A \otimes \neg(A \otimes \mathsf{force})"] \arrow[dd, "\cong"] & A' \otimes A \otimes \neg(A \otimes \neg\neg(A' \otimes \neg B)) \arrow[d, "A' \otimes \mathsf{apply}_A"] \\ & A' \otimes \neg(A' \otimes \neg B) \arrow[d, "\mathsf{apply}_{A'}"] \\ A \otimes A' \otimes \neg(A \otimes A' \otimes \neg B) \arrow[r, "\mathsf{apply}_{A \otimes A'}"] & B \end{tikzcd}$$

could equivalently be expressed in terms of coherence for the indexed negation

$$\begin{tikzcd} A' \otimes A \otimes \neg(A \otimes A' \otimes B) \arrow[r, "A' \otimes \neg_A \mathsf{id}_{A \otimes A' \otimes B}"] \arrow[d, "\cong"] & A' \otimes \neg(A' \otimes B) \arrow[d, "\neg_{A'} \mathsf{id}_{A' \otimes B}"] \\ A \otimes A' \otimes \neg(A \otimes A' \otimes B) \arrow[r, "\neg_{A \otimes A'} \mathsf{id}_{A \otimes A' \otimes B}"] & \neg B \end{tikzcd}$$

4.5 λ-abstraction in a $\otimes\neg$-category

Just as in the standard CPS transforms, function types $\sigma \to \tau$ will be decomposed into continuations for arguments σ and result continuations $\neg\tau$. So instead of exponentials, we have a derived notion of arrow type

$$[A \to B] \stackrel{\text{def}}{=} \neg(A \otimes \neg B)$$

The corresponding application map is the unit of the adjunction

$$\mathsf{apply}_A : A \otimes \neg(A \otimes \neg B) \longrightarrow B$$

In a cartesian closed category, we could define λ-abstraction in terms of the right adjoint $[A \to (\_)]$ in $A \times (\_) \dashv [A \to (\_)]$ and the unit of adjunction (the curried pairing map) $\mathsf{pair} : C \longrightarrow [A \longrightarrow (A \times C)]$ as $\lambda f \stackrel{\text{def}}{=} \mathsf{pair}; [A \to f]$. The notion of λ-abstraction that we have in the present setting can be defined in a way that is formally very similar, although we do not have cartesian closure.

We define a pairing map as

$$\mathsf{pair} \stackrel{\text{def}}{=} \mathsf{thunk}_C; \neg(A \otimes \neg(A \otimes \mathsf{force}); \mathsf{apply}_A) : C \longrightarrow \neg(A \otimes \neg(A \otimes C))$$

Note that of the two possible composites of the functors $A \otimes (\_)$ and $\neg$, one is self-adjoint on the left, the other on the right. $A \otimes \neg(\_)$ is self-adjoint on the left $\neg(A \otimes (\_))$ restricted to the centre is self-adjoint on the right.

The pairing map then allows us to define (call-by-value) λ-abstraction.

$$\overline{\lambda} f \stackrel{\text{def}}{=} \mathsf{pair}; \neg(A \otimes \neg f)$$

Although we may read apply and pair as having the types familiar from cartesian closed categories, that is

$$\begin{aligned} \mathsf{apply} &: A \otimes [A \to B] \longrightarrow B \\ \mathsf{pair} &: C \longrightarrow [A \to (A \otimes C)] \end{aligned}$$

this is really a kind of secondary etymology, as in reality apply and pair are the unit/counit of negation functors $A \otimes \neg(\_)$ and $\neg(A \otimes (\_))$, respectively.

$$\begin{aligned} \mathsf{apply} &: A \otimes \neg(A \otimes \neg B) \longrightarrow B \\ \mathsf{pair} &: C \longrightarrow \neg(A \otimes \neg(A \otimes C)) \end{aligned}$$

4.5.1 Proposition The following two diagrams commute

$$\begin{tikzcd} A \otimes C \arrow[r, "A\otimes\mathsf{pair}"] \arrow[dr, "\mathsf{id}_{A\otimes C}"'] & A \otimes \neg(A \otimes \neg(A \otimes C)) \arrow[d, "\mathsf{apply}"] \\ & A \otimes C \end{tikzcd}$$

$$\begin{tikzcd} \neg(A \otimes \neg B) \arrow[r, "\mathsf{pair}"] \arrow[dr, "\mathsf{id}_{\neg(A\otimes\neg B)}"'] & \neg(A \otimes \neg(A \otimes \neg(A \otimes \neg B))) \arrow[d, "\neg(A\otimes\neg\mathsf{apply})"] \\ & \neg(A \otimes \neg B) \end{tikzcd}$$

In the framework of premonoidal categories, a notion of call-by-value λ-abstraction was proposed [Pow]. Formally, this is a mild variation on monoidal closure, requiring not $A \otimes (\_)$, but it composition with the inclusion of the centre to have a right adjoint, hence the name "central" closure.

4.5.2 Proposition A $\otimes\neg$-category is centrally closed, in the sense that

$$\iota((\_) \otimes A) \dashv \neg(A \otimes \neg(\_))$$

where $\iota : \mathsf{Z}(\mathcal{K}) \hookrightarrow \mathcal{K}$. In terms of equations:

$$\begin{array}{rcl} A \otimes \overline{\lambda}(f); \mathsf{apply} & = & f \\ \overline{\lambda}(A \otimes g; \mathsf{apply}) & = & g \\ \overline{\lambda}(A \otimes g; f) & = & g; \overline{\lambda} f \\ \overline{\lambda} f; \neg(A \otimes \neg h) & = & \overline{\lambda}(f; h) \end{array}$$

where g is central.

The fact that pair is not natural with respect to all morphisms is what makes the "protecting by a λ" technique work in this setting. (It is really the precomposition with thunk that does the protecting.)

4.5.3 Corollary The inclusion of the centre is left adjoint to double negation.

$$\iota \dashv \neg\neg$$

The isomorphism of adjunction is

$$\begin{array}{rcl} \lambda_{\mathbf{1}} : \mathcal{K}(A, B) & \longrightarrow & \mathsf{Z}(\mathcal{K})(A, \neg\neg B) \\ f & \mapsto & \mathsf{thunk}; \neg\neg f \end{array}$$

This is a categorical formulation of thunking as a form of reification, given here by $f \mapsto \mathsf{thunk}; \neg\neg f$.

4.5.4 Corollary $\mathcal{K}$ is the Kleisli category for the monad $\neg\neg$ on $\mathsf{Z}(\mathcal{K})$.

We nonetheless prefer to regard the category of computations $\mathcal{K}$ as primary, rather than reifying everything and then running a Kleisli interpreter on top of it.

4.5.5 Proposition The call-by-name application and abstraction satisfy the following:

$$\begin{array}{rcl} A \otimes \underline{\underline{\lambda}}_A f; \underline{\underline{\mathsf{apply}}}_A & = & f \\ \underline{\underline{\lambda}}_A (A \otimes g; \underline{\underline{\mathsf{apply}}}_A) & = & g \end{array}$$

Moreover, $\underline{\underline{\mathsf{apply}}}_A$ is itself central.

It could appear as if we were somehow recovering a Cartesian closed category. But that is not really the case. Although the centre does have finite products, the centrally closed structure given by apply does not restrict to the centre, as apply is not itself central. Intuitively, this is because the application map is a jump with arguments, and jumps are too effectful to be central. So there is a trade-off of sorts: one can either have the products *or* the higher-order structure.

Chapter 5

The CPS term model

In this chapter, we build a term model of a $\otimes\neg$-category from the syntax of (simply-typed) CPS calculus. This is analogous to the (standard) construction of a Cartesian closed category from simply-typed λ-calculus (e.g. [Cro93]).

The CPS calculus is of course rather different in style from λ-calculi, so instead of familiar structures like (Cartesian) closure, or its generalisation to other binding constructs as adjoints to reindexing [1], we get self-adjointness as the algebraic manifestation of first-class jumping (hinted at in section 1.2.1 in the introduction).

5.1 Building a category from CPS terms

In this section we will attempt to isolate the crucial structure that makes continuation semantics work, gradually abstracting from the CPS calculus to arrive at a syntax free-presentation that will lead to a categorical semantics in the next section.

The jumping and the binding construct of CPS calculus correspond to identities and composition in the term model.

The self-adjointness on the left is responsible for various generalised jumps (which eliminates double-negations); while the self-adjointness on the right builds new places to jump to (which introduces a double negation).

5.1.1 First-order structure

In [App92], CPS terms are likened to the machine code of a von Neumann machine (as far as control is concerned). But as far as sequential composition is concerned, CPS terms are even more low-level than code, as they do not have a default or current continuation. The basic idea, then, is to ascribe meaning

[1] Consider for instance the definitions of $\prod$, $\sum$, $\forall$, $\exists$ in a topos

not to a CPS term by itself, but to a CPS typing judgement that lists the free variables of the term, distinguishing one of them as the current continuation.

The type environment part $\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau}$ of a judgement $\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M$ gives each morphism $[\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M]$ a unique domain and codomain, as required. Conversely, if the domain $\vec{\sigma}$ and codomain $\vec{\tau}$ are clear from the context, we can write more succinctly $[\vec{x}k \vdash M]$.

Once a *current continuation* has been singled out in a judgement, there is a natural concept of sequential composition:

$$[\vec{x}n \vdash M]; [\vec{y}k \vdash N] \stackrel{\text{def}}{=} [\vec{x}k \vdash M\{n\langle\vec{y}\rangle{=}N\}]$$

This has $[\vec{x}k \vdash k\langle\vec{x}\rangle]$ as its identity.

Furthermore, we have product types (by concatenation); and although these are not categorical products, we do have projections $\pi_i = [\vec{x}_1\vec{x}_2k \vdash k\langle\vec{x}_i\rangle]$ as well as contraction $\delta = [\vec{x}k \vdash k\langle\vec{x}\vec{x}\rangle]$.

Hence we have enough parallelism on types to accommodate multi-arity maps denoting program phrases of more than one free variable. Given the sequential nature of CPS, we would not expect genuine parallelism of morphisms (denoting terms) $f \otimes g$.

Given two CPS judgements

$$[\vec{x}k \vdash M] \qquad [\vec{y}h \vdash N]$$

there are in general two *different* composites; we can run either M or N first. In each case, the term first to be evaluated has to carry along the free variables needed by the second judgement. If we run M first, this gives a judgement $[\vec{x}\vec{y}k' \vdash M\{k\langle\vec{z}\rangle{=}k'\langle\vec{x}\vec{z}\rangle\}]$

But implicit in this composition of M and N was the notion of "carrying along free variables". Considered on its own this is, for every object A, a functor $A \otimes (\_)$ defined by

$$A \otimes [\vec{x}k \vdash M] \stackrel{\text{def}}{=} [\vec{y}\vec{x}k' \vdash M\{k\langle\vec{z}\rangle{=}k\langle\vec{y}\vec{z}\rangle\}]$$

Symmetrically, we have a functor $(\_) \otimes A$.

5.1.2 Application as double negation elimination

All the above are *trivial* terms in the sense that they jump to the current continuation. A CPS judgement that does not do this is evidently $[hk \vdash h\langle k\rangle]$. But this is just the identity $[kh \vdash h\langle k\rangle]$ with the argument and the current continuation interchanged.

More generally, given any $[hk \vdash M]$ with a continuation parameter h, let

$$\phi[hk \vdash M] \stackrel{\text{def}}{=} [kh \vdash M]$$

$\neg$ extends to a functor. We can define it in terms of ϕ and force as $\neg f = \phi(\mathsf{force}; f)$. Concretely, this boils down to the following:

$$\begin{aligned}
& \neg[\vec{x}h \vdash M] \\
= \ & \phi(\mathsf{force}; [\vec{x}h \vdash M]) \\
= \ & \phi([kl \vdash k\langle l\rangle]; [\vec{x}h \vdash M]) \\
= \ & \phi[kh \vdash k\langle l\rangle\{l\langle\vec{x}\rangle{=}M\}] \\
= \ & [hk \vdash k\langle l\rangle\{l\langle\vec{x}\rangle{=}M\}]
\end{aligned}$$

The switching operation does not interfere with any other names $\vec{x}$ in the environment, that is, a judgement $[\vec{x}hk \vdash M]$ can be switched to yield $[\vec{x}kh \vdash M]$. And this is natural.

This switching operation is quite unfamiliar from direct-style programming, but used on identities, projections and contraction it gives rise to some of the most important idioms that we need for the interpretation of λ-calculus with control.

force switching argument h and current continuation k in the identity $[hk \vdash k\langle h\rangle]$ gives

$$\mathsf{force} : [kh \vdash k\langle h\rangle] : \neg\neg A \longrightarrow A$$

apply switching argument f and current continuation k in the identity gives

$$\mathsf{apply} : [\vec{x}fk \vdash f\langle\vec{x}k\rangle] : A \otimes \neg(A \otimes \neg B) \longrightarrow B$$

throw switching h argument and current continuation k in the projection $\pi_1 = [\vec{x}hk \vdash k\langle\vec{x}\rangle]$ gives

$$\mathsf{throw} = [\vec{x}hk \vdash k\langle\vec{x}\rangle] : A \otimes \neg A \longrightarrow B$$

callcc switching argument k and current continuation f in $[kf \vdash f\langle kk\rangle]$ gives

$$[fk \vdash f\langle kk\rangle] : \neg(\neg A \otimes \neg A) \longrightarrow A$$

Note that the clauses for application, `throw` and `callcc` in figure 3.1.1 consist essentially of composing with one of these constants, while force is essentially the denotation of a free variable in call-by-name.

5.1.3 Thunking as double negation introduction

The units of the self-adjointness on the right wrap their argument into a thunk.

$$\mathsf{thunk} = [\vec{x}k \vdash h\langle f\rangle\{f\langle h\rangle{=}h\langle\vec{x}\rangle\}]$$

$$\mathsf{pair} = [\vec{x}k \vdash h\langle f\rangle\{f\langle\vec{y}h\rangle{=}h\langle\vec{y}\vec{x}\rangle\}]$$

5.2 The $\otimes\neg$ term model

5.2.1 Definition The category $\mathcal{K}(\mathrm{CPS})$ is constructed as follows. Objects are sequences $\vec{\tau}$ of types. A morphism from $\vec{\sigma}$ to $\vec{\tau}$ is an equivalence class $[\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M]$ of judgements, where $\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M$ and $\vec{x}'{:}\vec{\sigma}', k'{:}\neg\vec{\tau}' \vdash M'$ are equivalent iff

$$M = M'[\vec{x}'k' \mapsto \vec{x}k]$$

is derivable.

$$\begin{array}{rcl}\mathbf{Ob}\mathcal{K}(\mathrm{CPS}) & = & \{\tau_1 \ldots \tau_j \mid \tau_i \text{ is a CPS type expression}\}\\ \mathcal{K}(\mathrm{CPS})(\vec{\sigma},\vec{\tau}) & = & \{[\vec{x}:\vec{\sigma}, k:\neg\vec{\tau} \vdash M] \mid \vec{x}:\vec{\sigma}, k:\neg\vec{\tau} \vdash M \text{ is derivable}\}\end{array}$$

Identities and composition correspond to the two term-forming rules of CPS calculus.

$$\begin{array}{rcl}\mathsf{id}_{\vec{\sigma}} & = & [\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\sigma} \vdash k\langle\vec{x}\rangle]\\ [\vec{x}{:}\vec{\tau}_1, n{:}\neg\vec{\tau}_2 \vdash M]; [\vec{y}{:}\vec{\tau}_2, h{:}\neg\vec{\tau}_3 \vdash M] & = & [\vec{x}{:}\vec{\tau}_1, h{:}\neg\vec{\tau}_3 \vdash M\{n\langle\vec{y}\rangle{=}N\}]\end{array}$$

The structure on morphisms is given as follows:

$$\begin{array}{rcl}\neg[\vec{x}{:}\vec{\sigma}, h{:}\neg\vec{\tau} \vdash M] & = & [h{:}\neg\vec{\tau}, k{:}\neg\neg\vec{\sigma} \vdash k\langle f\rangle\{f\langle\vec{x}\rangle{=}M\}]\\ [\vec{x}{:}\vec{\sigma}_1, k{:}\neg\vec{\tau} \vdash M] \otimes \vec{\sigma}_2 & = & [\vec{x}{:}\vec{\sigma}_1, \vec{y}{:}\vec{\sigma}_2, h{:}\neg(\vec{\tau}\vec{\sigma}_2) \vdash M\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{y}\rangle\}]\\ \vec{\sigma}_1 \otimes [\vec{x}{:}\vec{\sigma}_2, k{:}\neg\vec{\tau} \vdash M] & = & [\vec{y}{:}\vec{\sigma}_1, \vec{x}{:}\vec{\sigma}_2, h{:}\neg(\vec{\sigma}_1\vec{\tau}) \vdash M\{k\langle\vec{z}\rangle{=}h\langle\vec{y}\vec{z}\rangle\}]\\ \mathsf{apply}_{\vec{\sigma}} & = & [\vec{x}{:}\vec{\sigma}, f{:}\neg(\vec{\sigma},\neg\vec{\tau}), k{:}\neg\vec{\tau} \vdash f\langle\vec{x}k\rangle]\\ \mathsf{thunk}_{\vec{\tau}} & = & [\vec{x}{:}\vec{\tau}, k{:}\neg\neg\neg\vec{\tau} \vdash k\langle f\rangle\{f\langle h\rangle{=}h\langle\vec{x}\rangle\}]\end{array}$$

5.2.2 Remark We should point out that focussing on the term model is not really a restriction to syntactic, as opposed to semantic models.

Consider the definition of the premonoidal structure on the category of continuation transformers on a Cartesian closed category, where a morphism is of the following form:

$$\Phi : R^A \longleftarrow R^B$$

It would be easy to say that $C \otimes \Phi$ is given by virtue of the functor $(\_)^C$ and the evident isomorphims

$$R^{C\times A} \cong (R^A)^C \longleftarrow (R^B)^C \cong R^{C\times B}$$

As soon as definitions become more complicated, that style becomes hopeless and one needs to adopt a more systematic approach by using the internal langauge of the Cartesian closed category, that is simply-typed λ-calculus with products and a constant Φ for every morphism Φ in the category. Then we can write the definition more concisely and rigorously as:

$$C \otimes \Phi \stackrel{\text{def}}{=} [\![k : R^{C\times A} \vdash \lambda(c,a) : C \times A.\Phi\,(\lambda b.k\,(c,b))\,a : R]\!]$$

Up to ordering of variables, this is essentially the definition in the CPS term model, with CPS specialised to the λ-calculus. Moreover, the axioms of the CPS calculus are sound for the translation to λ-calculus, hence proofs are translated to proofs about (the internal language of) Cartesian closed categories.

Reasoning within the CPS term model is thus similar to using the internal language of a Cartesian closed category or the internal logic of a topos instaed of doing diagram chases.

5.2.3 Proposition $\mathcal{K}(\mathrm{CPS})$ as defined in Definition 5.2.1 is in fact a category.

PROOF

- id is the identity

$$\begin{array}{rl} & \mathsf{id}; [\vec{y}h \vdash N] \\ = & [\vec{x}n \vdash n\langle\vec{x}\rangle]; [\vec{y}h \vdash N] \\ = & [\vec{x}h \vdash n\langle\vec{x}\rangle\{n\langle\vec{y}\rangle{=}N\}] \\ = & [\vec{x}h \vdash N[\vec{y} \mapsto \vec{x}]] \\ = & [\vec{y}h \vdash N] \end{array}$$

$$\begin{array}{rl} & [\vec{x}n \vdash M]; \mathsf{id} \\ = & [\vec{x}n \vdash M]; [\vec{y}h \vdash h\langle\vec{y}\rangle] \\ = & [\vec{x}h \vdash M\{n\langle\vec{y}\rangle{=}h\langle\vec{y}\rangle\}] \\ = & [\vec{x}h \vdash M[n \mapsto h]] \\ = & [\vec{x}n \vdash M] \end{array}$$

- composition is associative

$$\begin{array}{rl} & ([\vec{x}m \vdash L]; [\vec{y}n \vdash M])\,; [\vec{z}k \vdash N] \\ = & [\vec{x}n \vdash L\{m\langle\vec{y}\rangle{=}M\}]; [\vec{z}k \vdash N] \\ = & [\vec{x}k \vdash L\{m\langle\vec{y}\rangle{=}M\}\{n\langle\vec{z}\rangle{=}N\}] \\ = & [\vec{x}k \vdash L\{n\langle\vec{z}\rangle{=}N\}\{m\langle\vec{y}\rangle{=}M\{n\langle\vec{z}\rangle{=}M\}\}] \\ = & [\vec{x}k \vdash L\{m\langle\vec{y}\rangle{=}M\{n\langle\vec{z}\rangle{=}M\}\}] \\ = & [\vec{x}m \vdash L];\ ([\vec{y}n \vdash M]; [\vec{z}k \vdash N]) \end{array}$$

□

5.2.4 Lemma A morphism $[\vec{x}k \vdash M]$ is central iff for all N with $\vec{z} \notin \mathsf{FV}(N)$, $\vec{w} \notin \mathsf{FV}(M)$

$$M\{k\langle\vec{z}\rangle{=}N\{h\langle\vec{w}\rangle{=}l\langle\vec{z}\vec{w}\rangle\}\} = N\{h\langle\vec{w}\rangle{=}M\{k\langle\vec{z}\rangle{=}l\langle\vec{z}\vec{w}\rangle\}\}$$

PROOF Let $f = [\vec{x}k \vdash M] : A \longrightarrow A'$ be central. Then for all $g = [\vec{y}h \vdash N] : B \to B'$, $f \otimes B; A' \otimes g = A \otimes g; f \otimes B'$.

$$\begin{array}{rl} & f \otimes B; A' \otimes g \\ = & [\vec{x}\vec{y}l \vdash M\{k\langle\vec{z}\rangle{=}n\langle\vec{z}\vec{y}\rangle\{n\langle\vec{z}\vec{y}\rangle{=}N\{h\langle\vec{w}\rangle{=}l\langle\vec{z}\vec{w}\rangle\}\}\}] \\ = & [\vec{x}\vec{y}l \vdash M\{k\langle\vec{z}\rangle{=}N\{h\langle\vec{w}\rangle{=}l\langle\vec{z}\vec{w}\rangle\}\}] \end{array}$$

$$\begin{array}{rl} & A \otimes g; f \otimes B' \\ = & [\vec{x}\vec{y}l \vdash N\{h\langle\vec{w}\rangle{=}m\langle\vec{x}\vec{w}\rangle\{m\langle\vec{x}\vec{w}\rangle{=}M\{k\langle\vec{x}\rangle{=}\vec{z}\vec{w}\langle\}\rangle\}\}] \\ = & [\vec{x}\vec{y}l \vdash N\{h\langle\vec{w}\rangle{=}M\{k\langle\vec{z}\rangle{=}l\langle\vec{z}\vec{w}\rangle\}\}] \end{array}$$

$\square$

For instance, the identity $[\vec{x}k \vdash k\langle\vec{x}\rangle]$ is central, but $\mathsf{force} = [hk \vdash h\langle k\rangle]$ is not: take $M = x\langle k\rangle$ and $N = y\langle h\rangle$.

5.2.5 Lemma If a morphism f is central, then $\mathsf{thunk}; \neg\neg f = f; \mathsf{thunk}$.

PROOF If $f = [\vec{x}k \vdash M]$ is central, then by Lemma 5.2.4,

$$M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle l\rangle{=}z\langle\vec{y}l\rangle\}\} \quad = \quad h\langle f\rangle\{f\langle l\rangle{=}M\{k\langle\vec{y}\rangle{=}z\langle\vec{y}l\rangle\}\}$$

Hence, applying $\_\{z\langle\vec{y}l\rangle{=}l\langle\vec{y}\rangle\}$, we have

$$\begin{array}{rl} & M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle l\rangle{=}z\langle\vec{y}l\rangle\}\}\{z\langle\vec{y}l\rangle{=}l\langle\vec{y}\rangle\} \\ = & h\langle f\rangle\{f\langle l\rangle{=}M\{k\langle\vec{y}\rangle{=}z\langle\vec{y}l\rangle\}\}\{z\langle\vec{y}l\rangle{=}l\langle\vec{y}\rangle\} \end{array}$$

And this simplifies to

$$M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle l\rangle{=}l\langle\vec{y}\rangle\}\} = h\langle f\rangle\{f\langle k\rangle{=}M\} \tag{5.1}$$

Now

$$\begin{array}{rl} & \neg\neg f \\ = & \neg\neg[\vec{x}k \vdash M] \\ = & \neg[kp \vdash p\langle g\rangle\{g\langle\vec{x}\rangle{=}M\}] \\ = & [ph \vdash h\langle f\rangle\{f\langle k\rangle{=}p\langle g\rangle\{g\langle\vec{x}\rangle{=}M\}\}] \end{array}$$

$$\begin{array}{rl} & \mathsf{thunk}; \neg\neg f \\ = & [\vec{x}n \vdash n\langle p\rangle\{p\langle k\rangle{=}k\langle\vec{x}\rangle\}]; [ph \vdash h\langle f\rangle\{f\langle k\rangle{=}p\langle g\rangle\{g\langle\vec{x}\rangle{=}M\}\}] \\ = & [\vec{x}h \vdash n\langle p\rangle\{p\langle k\rangle{=}k\langle\vec{x}\rangle\}\{n\langle p\rangle{=}h\langle f\rangle\{f\langle k\rangle{=}p\langle g\rangle\{g\langle\vec{x}\rangle{=}M\}\}\}] \end{array}$$

$$\begin{aligned}
&= [\vec{x}h \vdash h\langle f\rangle\{f\langle k\rangle{=}p\langle g\rangle\{g\langle\vec{x}\rangle{=}M\}\}\{p\langle k\rangle{=}k\langle\vec{x}\rangle\}] \\
&= [\vec{x}h \vdash h\langle f\rangle\{f\langle k\rangle{=}g\langle\vec{x}\rangle\{g\langle\vec{x}\rangle{=}M\}\}] \\
&= [\vec{x}h \vdash h\langle f\rangle\{f\langle k\rangle{=}M\}]
\end{aligned}$$

$$\begin{aligned}
& f;\mathsf{thunk} \\
&= [\vec{x}k \vdash M];[\vec{y}h \vdash h\langle f\rangle\{f\langle l\rangle{=}l\langle\vec{y}\rangle\}] \\
&= [\vec{x}h \vdash M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle l\rangle{=}l\langle\vec{y}\rangle\}\}]
\end{aligned}$$

So by (5.1), $\mathsf{thunk};\neg\neg f = f;\mathsf{thunk}$.

□

This implies that thunk is a natural transformation in the centre.

5.2.6 Remark The isomorphism of adjunction of the self-adjointness on the right is a map $\psi : \hom(A, \neg B) \longrightarrow \hom(B, \neg A)$ defined by

$$\psi[\vec{x}k \vdash M] = [\vec{y}h \vdash k\langle f\rangle\{f\langle\vec{x}\rangle{=}M\{k\langle l\rangle{=}l\langle\vec{y}\rangle\}\}]$$

This is an involution when restricted to central morphisms; for general morphisms, it is not quite an involution:

$$\psi\psi[\vec{x}k \vdash M] = [\vec{x}k \vdash k\langle g\rangle\{g\langle\vec{y}\rangle{=}M\{k\langle l\rangle{=}l\langle\vec{y}\rangle\}\}]$$

Consider $\vec{x} = a$ and $M = a\langle k\rangle$.

5.2.7 Proposition The centre of $\mathcal{K}(\mathrm{CPS})$ has finite products: $\otimes$ together with the evident projections

$$\pi_j \stackrel{\mathrm{def}}{=} [\vec{x}_1\vec{x}_2k \vdash k\langle\vec{x}_j\rangle]$$

is a product in the centre of $\mathcal{K}(\mathrm{CPS})$; and the empty sequence together with evident morphism $[\vec{x}k \vdash k\langle\rangle]$ is a terminal object in the centre.

PROOF For $f_i = [\vec{x}k_i \vdash M_i]$, let

$$\langle f_1, f_2\rangle \stackrel{\mathrm{def}}{=} [\vec{x}h \vdash M_1\{k_1\langle\vec{y}_1\rangle{=}M_2\{k_2\langle\vec{y}_2\rangle{=}h\langle\vec{y}_1\vec{y}_2\rangle\}\}].$$

$$\begin{aligned}
& \langle\pi_1, \pi_2\rangle \\
&= \langle[\vec{x}_1k_1 \vdash k_1\langle\vec{x}_1\rangle], [\vec{x}_2k_2 \vdash k_2\langle\vec{x}_2\rangle]\rangle \\
&= [\vec{x}_1\vec{x}_2h \vdash k_1\langle\vec{x}_1\rangle\{k_1\langle\vec{x}_1\rangle{=}k_2\langle\vec{x}_2\rangle\{k_2\langle\vec{x}_2\rangle{=}h\langle\vec{x}_1\vec{x}_2\rangle\}\}] \\
&= [\vec{x}_1\vec{x}_2h \vdash h\langle\vec{x}_1\vec{x}_2\rangle] \\
&= \mathsf{id}
\end{aligned}$$

Note that central morphisms can be copied and discarded because the corresponding thunks can. This is the key to the proof, the remainder being routine manipulations of the definition of $\langle\_,|\rangle$. Let $[\vec{x}k \vdash M]$ be central. Then

$$M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle l\rangle{=}l\langle\vec{y}\rangle\}\} = h\langle f\rangle\{f\langle k\rangle{=}M\}$$

Therefore

$$\begin{aligned}
& M\{k\langle\vec{y}\rangle{=}l\langle\rangle\} \\
= \;& M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle k\rangle{=}k\langle\vec{y}\rangle\}\}\{h\langle f\rangle{=}l\langle\rangle\} \\
= \;& h\langle f\rangle\{f\langle k\rangle{=}M\}\{h\langle f\rangle{=}l\langle\rangle\} \\
= \;& l\langle\rangle\{f\langle k\rangle{=}M\} \\
= \;& l\langle\rangle
\end{aligned}$$

Similarly, we have

$$\begin{aligned}
& M\{k\langle\vec{y}\rangle{=}l\langle\vec{y}\vec{y}\rangle\} \\
= \;& M\{k\langle\vec{y}\rangle{=}p\langle\vec{y}\rangle\{p\langle\vec{y}_1\rangle{=}q\langle\vec{y}\rangle\{q\langle\vec{y}_2\rangle{=}l\langle\vec{y}_1\vec{y}_2\rangle\}\}\} \\
= \;& M\{k\langle\vec{y}\rangle{=}f\langle p\rangle\{p\langle\vec{y}_1\rangle{=}f\langle q\rangle\{q\langle\vec{y}_2\rangle{=}l\langle\vec{y}_1\vec{y}_2\rangle\}\}\{f\langle r\rangle{=}r\langle\vec{y}\rangle\}\} \\
= \;& M\{k\langle\vec{y}\rangle{=}h\langle f\rangle\{f\langle r\rangle{=}r\langle\vec{y}\rangle\}\}\{h\langle f\rangle{=}f\langle p\rangle\{p\langle\vec{y}_1\rangle{=}f\langle q\rangle\{q\langle\vec{y}_2\rangle{=}l\langle\vec{y}_1\vec{y}_2\rangle\}\}\} \\
= \;& h\langle f\rangle\{f\langle k\rangle{=}M\}\{h\langle f\rangle{=}f\langle p\rangle\{p\langle\vec{y}_1\rangle{=}f\langle q\rangle\{q\langle\vec{y}_2\rangle{=}l\langle\vec{y}_1\vec{y}_2\rangle\}\}\} \\
= \;& f\langle p\rangle\{p\langle\vec{y}_1\rangle{=}f\langle q\rangle\{q\langle\vec{y}_2\rangle{=}l\langle\vec{y}_1\vec{y}_2\rangle\}\}\{f\langle k\rangle{=}M\} \\
= \;& M\{k\langle\vec{y}_1\rangle{=}M\{k\langle\vec{y}_2\rangle{=}l\langle\vec{y}_1\vec{y}_2\rangle\}\}
\end{aligned}$$

This means that, for a central morphism g, we have $g;\langle\mathsf{id},\mathsf{id}\rangle = \langle g,g\rangle$. Let $g : A \longrightarrow B$ and $f_i : B \longrightarrow C_i$. Then

$$\begin{aligned}
& g;\langle f_1, f_2\rangle \\
= \;& g;\langle\mathsf{id},\mathsf{id}\rangle; B\otimes f_2; C_1 \otimes f_2 \\
= \;& \langle\mathsf{id},\mathsf{id}\rangle; A\otimes g; g\otimes B; B\otimes f_2; f_1\otimes C_2 \\
= \;& \langle\mathsf{id},\mathsf{id}\rangle; A\otimes g; A\otimes f_2; g\otimes C_2; f_1\otimes C_2 \\
= \;& \langle\mathsf{id},\mathsf{id}\rangle; A\otimes (g;f_2); (g;f_1)\otimes C_2 \\
= \;& \langle g;f_1, g;f_2\rangle
\end{aligned}$$

Therefore, for central g_i, $\langle g_1, g_2\rangle;\pi_j = g_j$ □

5.2.8 Lemma A morphism $f : A \longrightarrow \neg A'$ is central iff f is of the form

$$[\vec{x}k \vdash k\langle f\rangle\{f\langle\vec{y}\rangle{=}M'\}]$$

with $k \notin \mathsf{FV}(M')$.

PROOF Let $N = b\langle h\rangle$ with b and h fresh. Because $[\vec{x}k \vdash M]$ is central,

$$M\{k\langle a\rangle{=}b\langle h\rangle\{h\langle\vec{y}\rangle{=}l\langle a\vec{y}\rangle\}\} = b\langle h\rangle\{h\langle\vec{y}\rangle{=}M\{k\langle a\rangle{=}l\langle a\vec{y}\rangle\}\}$$

Applying $(\_)\{l\langle a\vec{y}\rangle{=}a\langle\vec{y}\rangle\}$ to both sides, we get

$$\begin{aligned} & M\{k\langle a\rangle{=}b\langle h\rangle\{h\langle\vec{y}\rangle{=}l\langle a\vec{y}\rangle\}\}\{l\langle a\vec{y}\rangle{=}a\langle\vec{y}\rangle\} \\ = \; & b\langle h\rangle\{h\langle\vec{y}\rangle{=}M\{k\langle a\rangle{=}l\langle a\vec{y}\rangle\}\}\{l\langle a\vec{y}\rangle{=}a\langle\vec{y}\rangle\} \end{aligned}$$

Simplifying this yields

$$M[k \mapsto b] = b\langle h\rangle\{h\langle\vec{y}\rangle{=}M\{k\langle a\rangle{=}a\langle\vec{y}\rangle\}\}$$

Hence

$$\begin{aligned} & f \\ = \; & [\vec{x}k \vdash M] \\ = \; & [\vec{x}b \vdash M[k \mapsto b]] \\ = \; & [\vec{x}b \vdash b\langle h\rangle\{h\langle\vec{y}\rangle{=}M\{k\langle a\rangle{=}a\langle\vec{y}\rangle\}\}] \\ = \; & [\vec{x}b \vdash b\langle h\rangle\{h\langle\vec{y}\rangle{=}M'\}] \end{aligned}$$

for $M' = M\{k\langle a\rangle{=}a\langle\vec{y}\rangle\}$ and $b \notin \mathsf{FV}(M')$, as b is fresh. □

5.2.9 Definition (Trivial CPS term) A CPS term is called *trivial in k* iff it is of the form

$$k\langle p_1 \dots p_n\rangle\{p_{i_1}\langle\vec{x}_{i_1}\rangle{=}M_{i_1}\}\dots\{p_{i_m}\langle\vec{x}_{i_m}\rangle{=}M_{i_m}\}$$

with $k \notin \mathsf{FV}(M_{i_j})$. $(\{i_1,\dots,i_m\} \subseteq \{1,\dots,n\})$

5.2.10 Proposition Suppose that there are no base types in CPS calculus. Then a morphism $f : A \longrightarrow B$ in $\mathcal{K}(\mathrm{CPS})$ is central iff $f = [\vec{x}k \vdash M]$ such that M is trivial in k.

This means we can find a trivial representative for the equivalence class, not that all representatives are of this form (one could simply expand a redex). For instance,

$$\mathsf{id} = [\vec{x}k \vdash k\langle\vec{x}\rangle] = [\vec{x}k \vdash n\langle h\rangle\{h\langle k\rangle{=}k\langle\vec{x}\rangle\}\{n\langle h\rangle{=}h\langle k\rangle\}] \;=\; \mathsf{thunk};\mathsf{force}$$

PROOF By cases on B. If $B = \neg\tau$, apply lemma 5.2.8. Otherwise, $B = \tau_0\tau_1\dots\tau_n$, and we proceed by induction on n. Since B is a sequence of type expressions, we split off the first one, which must be of the form $\tau_0 = \neg\sigma$. Apply the induction hypothesis to $f;\pi_2 : A \longrightarrow \tau_1 \ldots \tau_n$. By lemma 5.2.8, $f;\pi_1 : A \longrightarrow \tau_0 = \neg\sigma$, defined as

$$f;\pi_1 = [\vec{w}h \vdash M\{k\langle p\vec{y}\rangle{=}h\langle p\rangle\}]$$

must be of the form

$$f;\pi_1 = [\vec{w}h \vdash h\langle p\rangle\{p\langle\vec{x}\rangle{=}M'\}]$$

As f is central, $f = \langle f;\pi_1, f;\pi_2\rangle$. Then

$$\begin{array}{rl}
& \langle f;\pi_1, f;\pi_2\rangle \\
= & [\vec{w}k \vdash M\{k\langle p\vec{y}\rangle{=}h\langle p\rangle\{h\langle p\rangle{=}M\{k\langle q\vec{z}\rangle{=}k\langle p\vec{z}\rangle\}\}\}] \\
= & [\vec{w}k \vdash M\{k\langle p\vec{y}\rangle{=}h\langle p\rangle\}\{h\langle p\rangle{=}M\{k\langle q\vec{z}\rangle{=}k\langle p\vec{z}\rangle\}\}] \\
= & [\vec{w}k \vdash h\langle p\rangle\{p\langle\vec{x}\rangle{=}M'\}\{h\langle p\rangle{=}M\{k\langle q\vec{z}\rangle{=}k\langle p\vec{z}\rangle\}\}] \\
= & [\vec{w}k \vdash M\{k\langle q\vec{z}\rangle{=}k\langle p\vec{z}\rangle\}\{p\langle\vec{x}\rangle{=}M'\}] \\
= & [\vec{w}k \vdash k\langle pp_1 \ldots p_n\rangle\{p_{i_1}\langle\vec{x}_{i_1}\rangle{=}M_{i_1}\} \ldots \{p_{i_m}\langle\vec{x}_{i_m}\rangle{=}M_{i_m}\}\{p\langle\vec{x}\rangle{=}M'\}]
\end{array}$$

□

5.2.11 Conjecture We conjecture that respecting the finite product structure can be characterised syntactically by the occurrence of the current continuation.

- A morphism in $\mathcal{K}(\text{CPS})$ is cancellable iff it is of the form

$$[\vec{x}k \vdash k\langle p_1 \ldots p_l\rangle\{n_1\langle\vec{y}_1\rangle{=}N_1\} \ldots \{n_m\langle\vec{y}_m\rangle{=}N_m\}]$$

- A morphism in $\mathcal{K}(\text{CPS})$ is copyable iff it is of the form

$$[\vec{x}k \vdash q\langle p_1 \ldots p_l\rangle\{n_1\langle\vec{y}_1\rangle{=}N_1\} \ldots \{n_m\langle\vec{y}_m\rangle{=}N_m\}]$$

  with $k \notin \mathsf{FV}(N_j)$.

5.2.12 Proposition $\mathcal{K}(\text{CPS})$ is a $\otimes\neg$-category.

PROOF

- $\neg$ is functorial: $\neg$ preserves identities

$$\begin{array}{rl}
& \neg\mathsf{id}_{\vec{\tau}} \\
= & \neg[\vec{x}h \vdash h\langle\vec{x}\rangle] \\
= & [hk \vdash k\langle f\rangle\{f\langle\vec{x}\rangle{=}h\langle\vec{x}\rangle\}] \\
= & [hk \vdash k\langle h\rangle] \\
= & \mathsf{id}_{\neg\vec{\tau}}
\end{array}$$

$\neg$ preserves composition

$$\begin{aligned} & \neg[\vec{y}h \vdash N]; \neg[\vec{x}n \vdash M] \\ = \; & [hk' \vdash k'\langle n\rangle\{n\langle\vec{y}\rangle{=}N\}]; [nk \vdash k\langle m\rangle\{m\langle\vec{x}\rangle{=}M\}] \\ = \; & [hk \vdash k'\langle n\rangle\{n\langle\vec{y}\rangle{=}N\}\{k'\langle n\rangle{=}k\langle m\rangle\{m\langle\vec{x}\rangle{=}M\}\}] \\ = \; & [hk \vdash k\langle m\rangle\{m\langle\vec{x}\rangle{=}M\{n\langle\vec{y}\rangle{=}N\}\}] \\ = \; & \neg[\vec{x}h \vdash M\{n\langle\vec{y}\rangle{=}N\}] \\ = \; & \neg\,([\vec{x}n \vdash M]; [\vec{y}h \vdash N]) \end{aligned}$$

- $\phi_A : \mathcal{K}(A \otimes \neg B, C) \cong \mathcal{K}(A \otimes \neg C, B)$ is natural in B and C, i.e. $\phi_A(A \otimes \neg g; f) = \phi_A(f; g)$

$$\begin{aligned} & (\vec{\sigma} \otimes \neg[\vec{y}h' \vdash N])\,; \phi_{\vec{\sigma}}[\vec{x}kn \vdash M] \\ = \; & \vec{\sigma} \otimes [hk' \vdash k'\langle n\rangle\{n\langle\vec{y}\rangle{=}N\}]; [\vec{x}nk \vdash M] \\ = \; & [\vec{x}hh' \vdash k'\langle n\rangle\{n\langle\vec{y}\rangle{=}N\}\{k'\langle n\rangle{=}h'\langle\vec{x}n\rangle\}]; [\vec{x}nh \vdash M] \\ = \; & [\vec{x}hk \vdash k'\langle n\rangle\{n\langle\vec{y}\rangle{=}N\}\{k'\langle n\rangle{=}h'\langle\vec{x}n\rangle\}\{h'\langle\vec{x}n\rangle{=}M\}] \\ = \; & [\vec{x}hk \vdash h'\langle\vec{x}n\rangle\{n\langle\vec{y}\rangle{=}N\}\{h'\langle\vec{x}n\rangle{=}M\}] \\ = \; & [\vec{x}hk \vdash M\{n\langle\vec{y}\rangle{=}N\}] \\ = \; & \phi_{\vec{\sigma}}[\vec{x}kh \vdash M\{n\langle\vec{y}\rangle{=}N\}] \\ = \; & \phi_{\vec{\sigma}}\,([\vec{x}kn \vdash M]; [\vec{y}h \vdash N]) \end{aligned}$$

- $\phi_A : \mathcal{K}(A \otimes \neg B, C) \cong \mathcal{K}(A \otimes \neg C, B)$ is natural in A, i.e. $\phi_{A'}(g \otimes \neg B; f) = g \otimes \neg C; \phi_A(f)$

$$\begin{aligned} & \phi_{\vec{\sigma}}\,([\vec{x}n \vdash M] \otimes \neg\vec{\tau}; [\vec{y}hk \vdash N]) \\ = \; & \phi_{\vec{\sigma}}\,([\vec{x}hn' \vdash M\{n\langle\vec{y}\rangle{=}n'\langle\vec{y}h\rangle\}]; [\vec{y}hk \vdash N]) \\ = \; & \phi_{\vec{\sigma}}[\vec{x}hk \vdash M\{n\langle\vec{y}\rangle{=}n'\langle\vec{y}h\rangle\}\{n'\langle\vec{y}h\rangle{=}N\}] \\ = \; & [\vec{x}kh \vdash M\{n\langle\vec{y}\rangle{=}N\}] \\ = \; & [\vec{x}kh \vdash M\{n\langle\vec{y}\rangle{=}n'\langle\vec{y}k\rangle\}\{n'\langle\vec{y}k\rangle{=}N\}] \\ = \; & [\vec{x}kn' \vdash M\{n\langle\vec{y}\rangle{=}n'\langle\vec{y}k\rangle\}]; [\vec{y}kh \vdash N] \\ = \; & [\vec{x}n \vdash M] \otimes \neg\vec{\tau}'; \phi_{\vec{\sigma}}[\vec{y}hk \vdash N] \end{aligned}$$

- $(-) \otimes \vec{\sigma}$ is functorial

$$\begin{aligned} & \mathsf{id}_{\vec{\sigma}_1} \otimes \vec{\sigma}_2 \\ = \; & [\vec{x}k \vdash k\langle\vec{x}\rangle] \otimes \vec{\sigma} \\ = \; & [\vec{x}\vec{y}h \vdash k\langle\vec{x}\rangle\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{y}\rangle\}] \\ = \; & [\vec{x}\vec{y}h \vdash (h\langle\vec{z}\vec{y}\rangle)[\vec{z} \mapsto \vec{x}]] \\ = \; & [\vec{x}\vec{y}h \vdash h\langle\vec{x}\vec{y}\rangle] \\ = \; & \mathsf{id}_{\vec{\sigma}_1\vec{\sigma}_2} \\ = \; & \mathsf{id}_{\vec{\sigma}_1 \otimes \vec{\sigma}_2} \end{aligned}$$

$$\begin{aligned} & ([\vec{x}n \vdash M]; [\vec{y}k \vdash N]) \otimes \vec{\sigma} \\ = \; & [\vec{x}k \vdash M\{n\langle\vec{y}\rangle{=}N\}] \otimes \vec{\sigma} \\ = \; & [\vec{x}\vec{w}h \vdash M\{n\langle\vec{y}\rangle{=}N\}\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{w}\rangle\}] \\ = \; & [\vec{x}\vec{w}h \vdash M\{k\langle\vec{z}\rangle{=}h\langle\vec{x}\vec{z}\rangle\}\{n\langle\vec{y}\rangle{=}N\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{w}\rangle\}\}] \\ = \; & [\vec{x}\vec{w}h \vdash M\{n\langle\vec{y}\rangle{=}N\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{w}\rangle\}\}] \\ = \; & [\vec{x}\vec{w}k \vdash M\{n\langle\vec{y}\rangle{=}n'\langle\vec{y}\vec{w}\rangle\}\{n'\langle\vec{y}\vec{w}\rangle{=}N\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{w}\rangle\}\}] \\ = \; & [\vec{x}\vec{w}n' \vdash M\{n\langle\vec{y}\rangle{=}n'\langle\vec{y}\vec{w}\rangle\}]; [\vec{y}\vec{w}h \vdash N\{k\langle\vec{z}\rangle{=}h\langle\vec{z}\vec{w}\rangle\}] \\ = \; & [\vec{x}n \vdash M] \otimes \vec{\sigma}; [\vec{y}k \vdash M] \otimes \vec{\sigma} \end{aligned}$$

Analogously for $\vec{\sigma} \otimes (-)$.

$\Box$

In [Fil89], a different notion of value was proposed: a morphism $f : A \longrightarrow B$ is called total if it can be discarded in the sense that $A \longrightarrow B \longrightarrow \mathbf{1} = A \longrightarrow \mathbf{1}$. This is plausible insofar as jumps cannot be discarded, so that by excluding jumps one might hope to isolate those program phrases that do not have control effects.

5.2.13 Proposition $\otimes$ is not a product in the subcategory of total morphisms.

PROOF The following morphism twicecc is total (but not central). twicecc does not respect $\otimes$.

$$\mathsf{twicecc} \stackrel{\text{def}}{=} [\vec{x}hk \vdash k\langle\vec{x}l\rangle\{l\langle\vec{y}\rangle{=}k\langle\vec{y}h\rangle\}] : A \otimes \neg A \longrightarrow A \otimes \neg A$$

Informally, in terms of continuation transformers, twicecc could be read as "$k \mapsto k \circ k$".

This is total, because $\mathsf{twicecc}; [\vec{w}k \vdash k\langle\rangle] = [\vec{x}k \vdash k\langle\rangle]$ However, twicecc is not a value: it is too effectful to be copyable, in that $\mathsf{twicecc}; \langle\mathsf{id}, \mathsf{id}\rangle \neq \langle\mathsf{twicecc}, \mathsf{twicecc}\rangle$ can be distinguished by $\_; [zcfz'c'gr \vdash f\langle zq\rangle\{q\langle w\rangle{=}g\langle wc\rangle\}]$.

The two composites are "h^3" as distinct from "h^4", that is,

$$[\vec{x}ahk \vdash h\langle\vec{x}q\rangle\{q\langle\vec{w}\rangle{=}h\langle\vec{w}q\rangle\{q\langle\vec{w}\rangle{=}h\langle\vec{w}a\rangle\}\}]$$

$$[\vec{x}ahk \vdash h\langle\vec{x}q\rangle\{q\langle\vec{w}\rangle{=}h\langle\vec{w}q\rangle\{q\langle\vec{w}\rangle{=}h\langle\vec{w}q\rangle\{q\langle\vec{w}\rangle{=}h\langle\vec{w}a\rangle\}\}\}]$$

We omit the calculations here; this counterexample will be discussed in Chapter 6, where we demonstrate experimentally, by exhibiting programs, that twicecc cannot be copied. $\Box$

5.3 The indexed $\neg$ term model

We have mentioned in remark 4.4.1 during the discussion of $\otimes\neg$-categories that it may be helpful to think about negation as indexed.

As this chapter establishes the connection between CPS calculus and categories, we now sketch how this indexed viewpoint is related to a two-zone CPS calculus (see remark 2.4.3), in which operations such as negation affect only a subset of the environment. We arrive at an indexed category as a term model of the alternative (indexed) view of $\otimes\neg$-categories to be developed in chapter 8.

We define the equivalence of judgements as follows: $\vec{x}{:}\vec{\sigma}, k{:}\neg\vec{\tau} \vdash M$ and $\vec{x}'{:}\vec{\sigma}', k'{:}\neg\vec{\tau}' \vdash M'$ are equivalent iff

$$M = M'[\vec{x}'k' \mapsto \vec{x}k]$$

is derivable from the axioms of CPS calculus.

The indexed category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ is defined as follows

- the objects of $\mathcal{C}$ and all the fibres H_C are sequnces of CPS type expressions
- the product of objects in $\mathcal{C}$ is given by concatenation of sequences
- a morphism from $\vec{\sigma}$ to $\vec{\tau}$ in the base category $\mathcal{C}$ is an equivalence class of trivial judgements

$$\vec{y} : \vec{\sigma}, q : \neg\vec{\tau} \vdash q\langle p_1 \dots p_n\rangle\{p_{i_1}\langle\vec{x}_{i_1}\rangle{=}M_{i_1}\} \dots \{p_{i_m}\langle\vec{x}_{i_m}\rangle{=}M_{i_m}\}$$

- a morphism from $\vec{\tau}_1$ to $\vec{\tau}_2$ in $H\vec{\sigma}$ is an equivalence class of judgements

$$\vec{z} : \vec{\sigma}; \vec{x} : \vec{\tau}_1, k : \neg\vec{\tau}_2 \vdash M$$

- the structure on the fibres is given as follows

$$\begin{aligned}
\mathsf{id} &\stackrel{\mathrm{def}}{=} [\vec{z}; \vec{x}k \vdash k\langle\vec{x}\rangle] \\
[\vec{z}; \vec{x}n \vdash M]; [\vec{z}; \vec{y}k \vdash N] &\stackrel{\mathrm{def}}{=} [\vec{z}; \vec{x}k \vdash M\{n\langle\vec{y}\rangle{=}N\}] \\
\\
C^*[\vec{z}; \vec{y}k \vdash M] &\stackrel{\mathrm{def}}{=} [\vec{z}\vec{x}; \vec{y}k \vdash M] \\
L[\vec{z}\vec{x}; \vec{w}k \vdash M] &\stackrel{\mathrm{def}}{=} [\vec{z}; \vec{x}\vec{w}k' \vdash M\{k\langle\vec{y}\rangle{=}k'\langle\vec{x}\vec{y}\rangle\}] \\
\kappa_C[\vec{z}\vec{x}; \vec{w}k \vdash M] &\stackrel{\mathrm{def}}{=} [\vec{z}; \vec{x}\vec{w}k \vdash M] \\
\\
\neg[\vec{z}; \vec{x}k \vdash M] &\stackrel{\mathrm{def}}{=} [\vec{z}; kh \vdash h\langle f\rangle\{f\langle\vec{x}\rangle{=}M\}] \\
\mathsf{force} &\stackrel{\mathrm{def}}{=} [\vec{z}; hk \vdash h\langle k\rangle]
\end{aligned}$$

- Reindexing on f along h: for $f = [\vec{x}k \vdash M]$ and

$$h = [\vec{y}q \vdash q\langle p_1 \dots p_n\rangle\{p_{i_1}\langle\vec{x}_{i_1}\rangle{=}M_{i_1}\} \dots \{p_{i_m}\langle\vec{x}_{i_m}\rangle{=}M_{i_m}\}]$$

  let

$$H_h(f) \stackrel{\mathrm{def}}{=} [\vec{y}k \vdash M\{p_{i_1}\langle\vec{x}_{i_1}\rangle{=}M_{i_1}\} \dots \{p_{i_m}\langle\vec{x}_{i_m}\rangle{=}M_{i_m}\}]$$

5.4 Recursion in CPS

We can define a term model $\mathcal{K}$(RecCPS) of a $\otimes\neg$-category analogously to definition 5.2.1, but using the *recursive* CPS calculus.

Recursion plays no part in any of the categorical structure of a $\otimes\neg$-category, but we may ask what *additional* categorical structure it gives rise to. As a first step towards an answer to that, we sketch how a looping operator could be introduced categorically as a dinatural transformation. To some extent, this amount to a categorical account of "recursion from iteration" [Fil94a].

$$(\!|\mu f.\lambda x.M|\!)(k) \quad \stackrel{\text{def}}{=} \quad k\langle f\rangle\{f\langle\vec{x}h\rangle\Leftarrow(\!|M|\!)(h)\}$$

A looping operator on a $\otimes\neg$-category $\mathcal{K}$ is a dinatural transformation

$$\mathsf{fix}_A : \ \mathcal{K}(A\otimes(\_),(\_)) \xrightarrow{\ \cdot\cdot\ } \mathsf{Z}\mathcal{K}(A,\neg(\_))$$

$$\begin{tikzcd}
 & \mathcal{K}(A\otimes C,C) \arrow[r, "\mathsf{fix}_{A,C}"] & \mathsf{Z}\mathcal{K}(A,\neg C) \arrow[dr, "\mathsf{Z}\mathcal{K}(A,\neg C)"] & \\
\mathcal{K}(A\otimes B,C) \arrow[ur, "\mathcal{K}(A\otimes g,C)"] \arrow[dr, "\mathcal{K}(A\otimes B,g)"'] & & & \mathsf{Z}\mathcal{K}(A,\neg C) \\
 & \mathcal{K}(A\otimes B,B) \arrow[r, "\mathsf{fix}_{A,B}"] & \mathsf{Z}\mathcal{K}(A,\neg B) \arrow[ur, "\mathsf{Z}\mathcal{K}(A,\neg g)"'] &
\end{tikzcd}$$

For $A\otimes B \xrightarrow{f} C$ and $C \xrightarrow{g} B$

$$\mathsf{fix}_A(f;g);\neg g = \mathsf{fix}_A(A\otimes g;f)$$

5.4.1 Proposition In $\mathcal{K}=\mathcal{K}(\text{RecCPS})$, there is a looping operator

$$\mathsf{fix}_A : \ \mathcal{K}(A\otimes(\_),(\_)) \xrightarrow{\ \cdot\cdot\ } \mathsf{Z}(K)(A,\neg(\_))$$

given as follows

$$\mathsf{fix}_{\vec{\tau}}[\vec{x}:\vec{\tau},\vec{y}:\vec{\sigma},k:\neg\vec{\sigma}\vdash M] \quad \stackrel{\text{def}}{=} \quad [\vec{x}:\vec{\tau},h:\neg\neg\vec{\sigma}\vdash h\langle k\rangle\{k\langle\vec{y}\rangle\Leftarrow M\}]$$

Proof Let $f \stackrel{\text{def}}{=} [\vec{x}:\vec{\tau},\vec{y}n\vdash M]$ and $g \stackrel{\text{def}}{=} [\vec{z}k\vdash N]$. Then

$$\begin{aligned}
& \mathsf{fix}_{\vec{\tau}}(f;g);\neg g \\
= \ & \mathsf{fix}_{\vec{\tau}}[\vec{x}q\vdash h\langle k\rangle\{k\langle\vec{y}\rangle\Leftarrow M\{n\langle\vec{z}\rangle\Leftarrow N\}\}];[kq\vdash q\langle f\rangle\{f\langle\vec{z}\rangle\Leftarrow N\}] \\
= \ & [\vec{x}q\vdash h\langle k\rangle\{k\langle\vec{y}\rangle\Leftarrow M\{n\langle\vec{z}\rangle\Leftarrow N\}\}\{h\langle k\rangle\Leftarrow q\langle f\rangle\{f\langle\vec{z}\rangle\Leftarrow N\}\}] \\
= \ & [\vec{x}q\vdash q\langle f\rangle\{f\langle\vec{z}\rangle\Leftarrow N\}\{k\langle\vec{y}\rangle\Leftarrow M\{n\langle\vec{z}\rangle\Leftarrow N\}\}]
\end{aligned}$$

$$\begin{aligned}
& \mathsf{fix}_{\vec{\tau}}(\vec{\tau}\otimes g;f) \\
= \ & \mathsf{fix}_{\vec{\tau}}[\vec{x}\vec{z}n\vdash N\{k\langle\vec{y}\rangle\Leftarrow M\}] \\
= \ & [\vec{x}q\vdash q\langle n\rangle\{n\langle\vec{z}\rangle\Leftarrow N\{k\langle\vec{y}\rangle\Leftarrow M\}\}]
\end{aligned}$$

□

Our notion of equality for the recurive CPS calculus in not quite strong enough without some induction principle allowing us to conclude that

$$q\langle f\rangle\{f\langle\vec{z}\rangle{\Leftarrow}N\}\{k\langle\vec{y}\rangle{\Leftarrow}M\{n\langle\vec{z}\rangle{\Leftarrow}N\}\} = q\langle n\rangle\{n\langle\vec{z}\rangle{\Leftarrow}N\{k\langle\vec{y}\rangle{\Leftarrow}M\}\}$$

as required for the dinaturality. Alternatively, we could construct the term model from term modulo observational congruence 2.3.2. We conjecture that for this notion of equality the dinnaturality would follow, i.e.:

$$q\langle f\rangle\{f\langle\vec{z}\rangle{\Leftarrow}N\}\{k\langle\vec{y}\rangle{\Leftarrow}M\{n\langle\vec{z}\rangle{\Leftarrow}N\}\} \approx q\langle n\rangle\{n\langle\vec{z}\rangle{\Leftarrow}N\{k\langle\vec{y}\rangle{\Leftarrow}M\}\}$$

5.4.1 Recursion from iteration

The point of having a looping construct for recursively-defined continuations is that, given our decomposition of functions into special continuations, it is exactly what is needed for recursively-defined functions. The link is established by the self-adjointness (on the left).

If M is the body of a recursively defined function $f : \sigma \to \tau$ in the environment Γ, we have a judgement $\Gamma, x : \sigma, f : (\sigma \to \tau) \vdash M : \tau$ whose denotation is a morphism

$$[\![M]\!] : [\![\Gamma]\!] \otimes [\![\sigma]\!] \otimes \neg([\![\sigma]\!] \otimes \neg[\![\tau]\!]) \longrightarrow [\![\tau]\!]$$

applying the isomorphism of adjunction ϕ yields a morphism

$$\phi_{[\![\Gamma]\!]\otimes[\![\sigma]\!]}([\![M]\!]) : [\![\Gamma]\!] \otimes [\![\sigma]\!] \otimes \neg[\![\tau]\!] \longrightarrow [\![\sigma]\!] \otimes \neg[\![\tau]\!]$$

and looping this gives a morphism

$$[\![\Gamma]\!] \longrightarrow \neg([\![\sigma]\!] \otimes \neg[\![\tau]\!])$$

which is the denotation

$$[\![\mu f.\lambda x.M]\!] : [\![\Gamma]\!] \longrightarrow [\![\sigma \to \tau]\!]$$

This can be seen as a categorical distillation of Filinski's "Recursion from iteration" [Fil94a], where ϕ was called a "context-switch".

Chapter 6

Effects in the presence of first-class continuations

In this chapter, we demonstrate that first-class continuations give rise to strong and rather subtle effects. First of all, this is an illustration and validation of our categorical semantics. The issue that we wish to clarify and give support to is our choice of the subcategory of effect-free computations.

While semantic considerations form the backdrop, it is also possible to read this chapter as an exploration of a simple idea: that the current continuation can be used *twice*. Once it is established that this can indeed be done, essentially the same idea leads to counterexamples to at least three separate conjectures:

- Andrzej Filinski's view of the total morphisms as effect-free
- The idempotency hypothesis of Andrzej Filinski and Amr Sabry
- The decomposition of `force` (Felleisen's $\mathcal{C}$-operator), attempted by James Laird

Put more positvely, this shows that `callcc` is very expressive (in the sense of [Fel91]). The examples here seem to indicate that, intuitively or "morally", first-class continuations ought to be grouped together with state among the strong computational effects and not with much weaker effects like divergence.

Preliminaries

We make use of the categorical combinators that were defined in figure 1.12 on page 19 and figure 1.13 on page 20 for ML and Scheme, respectively.

We need to make a distinction between jumps in which the value `thrown` is not itself of continuation (or function) type, as in a plain `goto` like `throw k ()`, and the unrestricted jumps afforded by `callcc`. We call the former first-order jump or exit, and the latter first-class jump.

6.1 Using the current continuation twice

One way of thinking about the denotations of terms in the presence of first-class continuations is as "continuation transformers" transforming a continuation for their codomain backwards into a continuation for their domain, by analogy with predicate transformers transforming postconditions into preconditions. At first sight this appears to be dual to the usual functional way of thinking about denotations as transforming a value (of the type given by the domain of the denotation) forward into a result (of the type given by the codomain). However, this is not really a duality, because, although each value transformer gives rise to a continuation transformer by precomposition, not every continuation transformer arises this way. Non-standard manipulations of the control flow, as by control operators, do not simple apply the current continuation to a result. For instance, a jump typically ignores the current continuation. (Strictly speaking, this applies only to first-order jumps that do not pass the current continuation as an argument the way `force` does.) But there are other ways, apart from applying it or ignoring it, of transforming the current continuation: such as using it *twice*.

The identification of functions with certain continuations cuts both ways: not only can we reduce functions to continuations; we may also regard continuations of the appropriate type as functions and treat them accordingly.

In the context of the present discussion, this means that we can regard a computation of type

```
'1a * '1a cont -> '1a * '1a cont
```

viewed (on the meta-level) as a continuation transformer

```
('1a * '1a cont) cont  -> ('1a * '1a cont) cont
```

as just a "function transformer" mapping the function space `'1a -> 'a` into itself. A fairly obvious candidate for such a function transformer is the function `twice` : $f \mapsto f \circ f$.

This continuation transformer counterpart of `twice`, while not representing a jump (ignoring the current continuation), is still a non-standard control manipulation, as it is different from applying the current continuation to a result.

6.1.1 Writing `twicecc` compositionally

We have two alternative ways of writing `twicecc`. We can take the CPS term

$$k\langle \vec{x} l\rangle\{l\langle \vec{y}\rangle{=}k\langle \vec{y} h\rangle\}$$

```
fun twicecc (n,h) = callcc(fn k =>
                      (fn n => throw k (n,h))
                      (callcc(fn q => throw k (n,q))));

    twicecc : '1a * '1a cont -> '1a * '1a cont;
```

Figure 6.1: `twicecc` in continuation-grabbing style (NJ-SML)

```
(define twicecc
  (lambda (l)
    (call/cc (lambda (f)
               ((lambda (n)
                  (f (list n (cadr l))))
                (call/cc (lambda (q)
                           (f (list (car l) q)))))))))
```

Figure 6.2: `twicecc` in continuation-grabbing style (Scheme)

```
fun twice f = f o f;

    twice : ('a -> 'a) -> ('a -> 'a);

fun twicecc a = (phi(funtocont o twice o conttofun) o thunk) a;

    twicecc :  '1a * '1a cont -> '1a * '1a cont;
```

Figure 6.3: `twicecc` in compositional style (NJ-SML)

```
(define (twice f)
  (compose f f))

(define twicecc
  (compose
   (phi
    (compose fun-to-cont twice cont-to-fun))
   thunk))
```

Figure 6.4: `twicecc` in compositional style (Scheme)

and do a continuation-grabbing style transform (see definition 2.6.8 on page 41) to arrive at an ML (and similarly, Scheme) program (figures 6.1 and 6.2).

This amounts to writing a continuation-passing style function composition in the source language (ML or Scheme), bypassing its control structure in favour of explicit jumps and continuation bindings. Although this method provides a practical use for continuation-grabbing style, it is somewhat rough and ready, in that it is a functional analogue of spaghetti coding and makes it harder for the compiler to supply useful type information for those subterms that are non-returning.

Informally, we could paraphrase figures 6.1 and 6.2 as follows. The current continuation is seized by a `callcc`. It is then treated as a function by being composed with itself. This composition, though, is done in the style of CPS. That is to say, it the continuation treated as a function is invoked twice, each time with an argument and a result continuation. Composition is achieved by making the result continuation of the first invocation to be evaluated (textually this is the second one) refer to the place where the second one expects its argument. This is done by the inner `callcc` seizing the λ in the operator position as its continuation.

A more structured approach would be to start with the familiar function $\mathsf{twice} \stackrel{\text{def}}{=} \lambda f.f \circ f$

$$\mathsf{twice} : [A \to A] \longrightarrow [A \to A]$$

which up to coercion is a map

$$\neg(A \otimes \neg A) \longrightarrow \neg(A \otimes \neg A)$$

Negating this and and composing with thunk and force yields

$$A \otimes \neg A \longrightarrow \neg\neg(A \otimes \neg A) \longrightarrow \neg\neg(A \otimes \neg A) \longrightarrow A \otimes \neg A$$

And this is what we do in figures 6.3 and 6.4, using the categorical combinators from figures 1.12 and 1.13. `twicecc` becomes a one-liner in ML, consisting mainly of function composition (with a η-redex to prevent non-generic weak type variable).

As a first illustration of what `twicecc` does, consider the following example in ML:

```
callcc(fn k =>
       (fn (n, h) => throw h (n+1))
       (twicecc (0, k)));
(* val it = 2 : int *)
```

Here the continuation of `twicecc` could be phrased as "pass the first argument incremented by one to the second argument". The continuation that `twicecc` supplies to its arguments, then, is *twice* that; hence 0 is incremented twice before finally being passed to the (top-level) continuation supplied by the surrounding `callcc`.

6.2 Copying and discarding

The fact that `twicecc` is total in the sense of discardable is corroborated by considering its composite with a function that discards its argument.

```
fun bang _ = ();
    bang : '1a -> unit;

fun discardtester testee =
callcc(fn k => ((fn _ => 42) o testee)(0,k));
```

We demonstrate the fact that `twicecc` is not copyable (see proposition 5.2.13) by counterexample.

Copying `twicecc` (using `copy_twicecc`) and attempting to copy its result after it has been run (using `twicecc_copy` produce different results:

```
- distinguisher copy_twicecc;
val it = 3 : int
- distinguisher twicecc_copy;
val it = 4 : int
```

The context that can distinguish `copy_twicecc` and `twicecc_copy`, abstracted as `distinguisher` above, could be visualised as follows.

$$(\underbrace{I}_{\mathtt{0}}\otimes\underbrace{\neg I}_{\mathtt{k}})\otimes\underbrace{\neg(I\otimes\neg I)}_{\mathtt{inc}}\to((\underbrace{I}_{\mathtt{n}}\otimes\underbrace{\neg I}_{\mathtt{h}})\otimes\underbrace{\neg(I\otimes\neg I)}_{\mathtt{f}})\otimes((I\otimes\neg I)\otimes\underbrace{\neg(I\otimes\neg I)}_{\mathtt{g}})$$

We can show similarly that `force` is not copyable either (figures 6.2 and 6.8). `force` is in some sense maximally effectful: it is a jump, but as it passes the current continuation as an argument, it is more sensitive than an ordinary jump, which is oblivious to its current continuation.

The distinguishing context to show that `force` is not copyable is built using the one for twicecc. Roughly speaking, `(force, force)` will behave like `(twicecc,twicecc)` when each occurrence of `force` is given a separate copy of `twicecc` wrapped in a thunk.

```
fun copy_twicecc x = (twicecc x, twicecc x);

fun twicecc_copy x = (fn y => (y,y)) (twicecc x);

fun distinguisher testee =
    callcc(fn k =>
           (fn (((n,h),f),(_,g)) =>
            throw h (conttofun f (conttofun g  n)))
           (testee ((0,k),funtocont (fn n => n+1))));
```

Figure 6.5: Effectfulness of `twicecc`. Copying a computation, copying its result and a context to distinguish them (NJ-SML)

```
(define copy-twicecc
  (lambda (l)
      (list (twicecc l) (twicecc l))))

(define twicecc-copy
  (lambda (l)
    ((lambda (y) (list y y))
     (twicecc l))))

(define distinguisher
  (lambda (testee)
    (call/cc (lambda (k)
               ((lambda (l)
                  ((cadaar l)
                   ((cont-to-fun (cadadr l))
                    ((cont-to-fun (cadar l))
                     (caaar l)))))
               (testee
                (list (list 0 k)
                      (fun-to-cont
                       (lambda (n) (+ n 1)))))))))
```

Figure 6.6: Effectfulness of `twicecc`. Copying a computation, copying its result and a context to distinguish them (Scheme)

```
fun copyforce h = (force h, force h);

fun forcecopy h = (fn a => (a,a)) (force h);

fun distinguisher2 f =  distinguisher (f o (delay twicecc));
```

Figure 6.7: `force` is not copyable (NJ-SML)

```
(define copy-force
  (lambda (h)
    (list (call/cc h) (call/cc h))))

(define force-copy
  (lambda (h)
    ((lambda (y) (list y y))
     (call/cc h))))

(define (distinguisher2 testee)
  (distinguisher
   (compose
    testee
    (negate (negate twicecc))
    thunk)))
```

Figure 6.8: force = `call/cc` is not copyable (Scheme)

6.2.1 `twicecc` is not thunkable

This also provides an example for the fact that `twicecc` is not thunkable in the sense that composing with `thunk` does not succeed in wrapping the computation of `twicecc` into a thunk. Hence the following simple-minded attempt to define the distinguishing context for the non-copyability of `force` does not work:

```
fun forcecopytesterwrong f =
    distinguisher (f o (pseudodelay twicecc));
```

Because `forcecopytesterwrong` cannot pass `twicecc` to each occurrence of `force`, both tests evaluate to the same value, 4.

```
- forcecopytesterwrong copyforce;
val it = 4 : int
- forcecopytesterwrong forcecopy;
val it = 4 : int
```

A proper distinguishing context uses the delaying idiom , which *negates* `twicecc`.

```
fun forcecopytester f =
    distinguisher (f o (delay twicecc));
```

Now the non-copyablity of `force` manifests itself in the same way as for `twicecc`

```
- forcecopytester copyforce;
val it = 3 : int
- forcecopytester forcecopy;
val it = 4 : int
```

6.2.2 Cancellable and copyable are orthogonal

Considering that values are copyable and discardable whereas jumps (`throw`) are copyable but not discardable, we can summarise that copyable and discardable are orthogonal.

While it is evident that values can be discarded and jumps cannot, the right column was previously thought to be unoccupied, in that Filinski [Fil89] thought that cancellability, separating the top from the bottom row, was sufficient for separating value from *all* effects.

	copyable	not copyable
discardable	x $\lambda x.M$	`twicecc a`
not discardable	`throw k 42`	`force h`

A corollary of this table is that a first-order jump like `throw 42` is not maximally effectful. When it comes to being effectful, it is self-defeating in that it forgets the current continuation. This implies that is it copyable, because one jump $(\lambda x.(x,x))(\mathtt{throw}\,k\,42)$ is as good as two $(\mathtt{throw}\,k\,42, \mathtt{throw}\,k\,42)$, because the first jump will ignore it continuation containing the second jump, so that it does not matter whether the latter is present or not. The quintessential first-*class* jump `force`, by contrast, is not oblivious to its continuation, as this is passed as an argument. This makes `force` sufficiently sensitive to its continuation to resist copying.

6.2.3 First-class control is not an idempotent effect

`distinguisher` also gives us a counterexample to the conjecture, due to Andrzej Filinski and Amr Sabry, that control is an idempotent effect; thanks to Andrzej Filinski for pointing this out to me. [Andrzej Filinski, personal communication].

The idempotency conjecture holds that $(\lambda x.(x,x))M$ should be indistinguishable from (M, M)

The conjecture could be perhaps be supported by informal arguments about first-order jumps. We have mentioned that these can be copied essentially because they are oblivious to their continuation, so that it does not matter if another jump follows. Hence the idempotency could be defended for values, as well as for first-order jumps. What it fails to take into account are terms that do not simply pass something to the current continuation, but do not ignore it either. There seems to be an assumption of a kind of excluded middle here, along the line of: functions in continuations semantics can return a value or else they are `goto`'s.

As witnessed by `twicecc`, first-class continuations are more subtle than that. Continuations of the appropriate type can be used just as ordinary function declared with a `fun` or `define`.

To refute the idempotency hypothesis, we once again use `distinguisher` and `twicecc`:

```
distinguisher (fn x => (fn y => (y,y)) (twicecc x));

distinguisher (fn x => (fn a => (fn b => (a,b))
              (twicecc x)) (twicecc x));
```

To the extent that that such ideological conclusions can be drawn from this example, we should like to argue that it is misleading to think of first-class control as a form of non-termination due to jumping.

6.3 Centrality and effect-freeness

Having demonstrated in section 6.2 that discardable morphisms are too permissive a notion to be a suitable characterization of effect-free computation, we now try to add some plausibility to the claim that centrality in the presence of first-class continuations *is* a suitable notion.

We mentioned in remark 4.4.2 that it is due to the self-adjointness that centrality can be assumed to imply effect-freeness. There is some room for misunderstanding here, as there is a different, but weaker, argument for such an implication. We hope to clarify the connection between centrality and effect-freeness *in the presence of first-class continuations* by some concrete examples

First note that we can talk about centrality in quite a general setting: whenever we have a language having a `let`- and a tuple construct, we can define a term M to be central iff for all fresh variables a and b and all other terms N,

$$\texttt{let } a = M \texttt{ in let } b = N \texttt{ in } (a, b)$$

is the same (under whatever notion of equality we happen to have) as

$$\texttt{let } b = N \texttt{ in let } a = M \texttt{ in } (a, b)$$

For instance, if our notion of effect is given by (not necessarily first-class) continuations and at least two different values that can be thrown, then terms M that throw cannot be central. We only need to take for N a term that throws something else in order to tell the difference between the two composites.

```
-(callcc (fn k =>
let val a = throw k "A side effect." in
let val b = throw k "A subtly different side effect.\n" in
        a  end end));
= = = val it = "A side effect." : string
- (callcc (fn k =>
let val b = throw k  "A subtly different side effect." in
let val a = throw k  "A side effect." in
        a  end end));
= = = val it = "A subtly different side effect." : string
```

If additional side-effects, such as input-output, are present in the language, it is quite straightforward to see that `twicecc` is not central; see figure 6.11. Some more care is needed if control is the only effect.

```
fun forcefirst (a,b) =
    let val y = force b in
    let val x =  (output(std_out, "A side effect.\n"); 42) in
        (x,y) end end;

fun forcelast (a,b) =
    let val x =  (output(std_out, "A side effect.\n"); 42) in
    let val y = force b in
        (x,y) end end;

fun trytoreify f (n,k) =
    (phi(fn h  =>
         (fn (x,y) => throw y x)
         (f ((n,k),h)))) (thunk ());

val effectnotinclosure = trytoreify forcelast ((),());
    effectnotinclosure  : int cont cont;


val effectinclosure = trytoreify forcefirst ((),());
    effectinclosure  : int cont cont;

force effectnotinclosure;

force effectinclosure;
```

Figure 6.9: force can reify by being precomposed (in ML)

```
(define (forcelast l)
  (let*
      ((x (begin (write "A side effect.") (newline) 42))
       (y (call/cc (cadr l))))
    (list x y)))

(define (forcefirst l)
  (let*
      ((y (call/cc (cadr l)))
       (x (begin (write "A side effect.") (newline) 42)))
    (list x y)))

(define (trytoreify f)
  (lambda (l)
    ((phi
      (lambda (h)
        ((lambda (l) ((cadr l) (car l)))
         (f (list l h)))))
     (thunk (list)))))

(define effectnotinclosure ((trytoreify forcelast) (list)))

(define effectinclosure ((trytoreify forcefirst) (list)))

(call/cc effectnotinclosure)

(call/cc effectinclosure)
```

Figure 6.10: force can reify by being precomposed (in Scheme)

However, with first-class continuations, one can do much more than subject M to testing for effects; one can actually reify M. For $N = \texttt{force}\ h$, the composite with `force` coming after M

$$\texttt{let}\ a = M\ \texttt{in}\ \texttt{let}\ b = \texttt{force}\ h\ \texttt{in}\ (a, b)$$

passes to h the continuation *after* running M; this gives access to the value that M returns after being run and possibly side-effecting. The composite with `force` coming first, by contrast, passes to h the continuation *before* M is computed. This has the same effect as wrapping the whole computation, include possible side-effects, into a thunk.

So instead of the somewhat weak argument "if M had effects, we should be able to find a test N that can tell the difference", we know that `force` will reify anything that follows. Now, intuitively speaking, in order for the two composites to agree, (i.e. for M thunked and unthunked to be the same) M itself must already be as good as reified.

Depending on whether `force` appears first or not, one can achieve either the genuine thunking

$$A \xrightarrow{\mathsf{thunk}} \neg\neg A \xrightarrow{\neg\neg f} \neg\neg B$$

or the "pseudo-delaying"

$$A \xrightarrow{f} B \xrightarrow{\mathsf{thunk}} \neg\neg B$$

The difference between these two is demonstrated in figures 6.9 and 6.10.

6.3.1 `twicecc` is not central

We established in 6.2 that `twicecc` cannot be copied. In our semantics, central morphisms respect the product and can be copied. Thus `twicecc` cannot be central — at least that is what the semantics predicts.

To show that this is indeed the case, and so to validate our semantics, we consider a final experiment.

As in the above, counterexamples are easier to find if we allow ourselves the additional observations afforded by I/O) — see figure 6.11 for a demonstration that `twicecc` is not central.

For the general case, without relying on I/O, we reuse the distinguishing context once more. The fact that `twicecc` is not central is demonstrated in figures 6.12 and 6.13.

```
callcc(fn k =>
       (fn (((),x),y) => throw x y)
        (let val y = output(std_out, "Side effect.\n") in
         let val x = twicecc ((),k) in
             (x,y) end end));

(* prints once *)

callcc(fn k =>
       (fn (((),x),y) => throw x y)
       (let val x = twicecc ((),k) in
        let val y = output(std_out, "Side effect.\n") in
            (x,y) end end));

(* prints twice *)
```

Figure 6.11: `twicecc` is not central (shown using I/O)

```
fun twicecc_first (a,b) =
       let val x = twicecc a in
       let val y = force b in
           (x,y) end end;

fun twicecc_last (a,b) =
       let val y = force b in
       let val x = twicecc a in
           (x,y) end end;

fun centralitytester testee =
    distinguisher ((fn h => (force h, force h))
                   o (trytoreify testee));

centralitytester twicecc_last;  (* val it = 3 : int *)

centralitytester twicecc_first; (* val it = 4 : int *)
```

Figure 6.12: `twicecc` is not central (In ML)

```
(define (twicecc_first l)
  (let*
      ((x (twicecc (car l)))
       (y (call/cc (cadr l))))
    (list x y)))

(define (twicecc_last l)
  (let*
     ((y (call/cc (cadr l)))
      (x (twicecc (car l))))
    (list x y)))

(define (centralitytester testee)
  (distinguisher
   (compose
    (lambda (h) (list (call/cc h) (call/cc h)))
    (trytoreify testee))))

(centralitytester twicecc_last)

(centralitytester twicecc_first)
```

Figure 6.13: `twicecc` is not central (In Scheme)

6.4 Another non-copyability result

We consider another morphism that is total, but not copyable, while easier to understand in intuituive terms than `twicecc`. We will also need it for the construction of counterexamples in Sections 6.5 and 6.7 below.

Written as a CPS term, twicecc seems to be the simples way of using the current continuation twice.

$$[\vec{x}hk \vdash k\langle xq\rangle\{q\langle\vec{y}\rangle{=}k\langle\vec{y}h\rangle\}]$$

If we want something of function type using its current continuation twice, we can write the following CPS term in the same spirit as twicecc, though slightly longer:

$$[k \vdash k\langle f\rangle\{f\langle xp\rangle{=}k\langle f\rangle\{f\langle yq\rangle{=}q\langle x\rangle\}\}]$$

Despite being longer as a CPS term, this is easier to write in ML or Scheme, requiring no messy Continuation Grabbing Style:

```
callcc(fn k => throw k (fn x => throw k (fn y => x)))}
```

This term passes a function to its current continuation `k`. When this function is called with an argument `x`, the constant function always returning that argument is passed to `k`. Hence the function eventually (on the second call to the current continuation) returned by the term is the function always returning the argument to the first call. We can regard this as the solution to the following continuation programming exercise:

> Define a function f such that all calls to f return the argument of the first call of f. Do not use state.

(We name this `argfc`, for "argument of first call".)

At first sight, it seems hard to see how to do this without state: the obvious solution, after all, uses two variables (or in ML, references): a non-local variable to hold the argument of the first invocation and a boolean flag to record if the function has been called before (if not, then the variable needs to be assigned). See figures 6.16 and 6.17 for a version of `argfc` with local state. In Scheme (figure 6.17), we can give a better analogue of `argfc` with continuations (figure 6.15) by using a function that updates its own definition when it is called.

```
fun argfc () =
    callcc(fn k =>
           throw k (fn x => throw k (fn y => x)));

let val f = argfc () in
    [f 1, f 2, f 3, f 4] end;

let val f = argfc () in
    [f 42, f 2, f 3, f 4] end;

fun distinguisher testee =
    let val (f, g) = testee argfc () in
        (f 1, g 2)
        end;

distinguisher (fn f => fn x => (f x, f x));
(* (1,2) : int * int *)

distinguisher (fn f => fn x => ((fn y => (y,y)) (f x)));
(*  (1,1) : int * int *)
```

Figure 6.14: `argfc` cannot be copied (in ML)

```
(define argfc
  (lambda ()
    (call/cc
     (lambda (k)
       (k
        (lambda (x)
          (k
           (lambda (y) x))))))))


(let ((f (argfc)))
  (list (f 1) (f 2) (f 3) (f 4)))

; a list with the all entries the same (1,2,3 or 4); unspecified which
; bigloo picks the last

(let ((f (argfc)))
  (list (f 5647) (f 3425) (f 2484) (f 75473)))

; Cannot be copied

(define (distinguisher testee)
  ((lambda (l)
     ((car l) 1)
     ((cadr l) 2))
   ((testee argfc))))

(distinguisher
 (lambda (f)
   (lambda ()
     (list (f) (f)))))              ;  2

(distinguisher
 (lambda (f)
   (lambda ()
     ((lambda (y) (list y y))       ;  1
      (f)))))
```

Figure 6.15: `argfc` cannot be copied (in Scheme)

```
fun argfc () =
    let val fc = ref true
        and arg = ref 0
    in
        fn x =>
        (if !fc then (fc := false; arg := x)
         else ();
             !arg)
    end;
```

Figure 6.16: `argfc` with local state (in ML)

```
; use local variable for argument of first call

(define (argfc)
  (let
      ((fc #t)
       (arg 0))
    (lambda (x)
      (if fc
          (begin
            (set! fc #f)
            (set! arg x)))
      arg)))

; use variable for the function

(define (argfc)
  (letrec
      ((fc #t)
       (f (lambda (x)
            (if fc
                (begin
                  (set! fc #f)
                  (set! f (lambda (y) x))))
            (f x))))
    f))
```

Figure 6.17: `argfc` with local state (in Scheme)

```
datatype void = VOID of void;

fun invoid (VOID x) =  invoid x;

fun callcc' f = callcc(fn k => f (fn x => VOID(throw k x)));
    callcc' :  (('1a -> void) -> '1a) -> '1a;
```

Figure 6.18: Variant of `callcc` with void-returning continuations

6.5 The failure of Laird's bootstrapping of `force`

In [Lai97], James Laird claims that control operators at ground type are sufficient in that one can inductively define them at function types. Specifically, he gives an inductive definition for the double-negation control operator of type

```
((('a -> '2b) -> void) -> void) -> 'a -> '2b;
```

in terms of that of type

```
(('2b -> void) -> void)  -> '2b)
```

[James Laird, personal communication/email].

For ML, the inductive definition supposes a variant of `callcc` in which continuations are identified with functions $\ldots -> $ `void`; see figure 6.18.

In figure 6.19, we give a simplistic version `laird1` first; the function `laird` can then be seen as a refinement thereof designed to cope with control effects in its argument. The Scheme analogue is in figure 6.20.

The informal argument for the correctness of this construction seems essentially similar to that which one could advance in favour of the idempotency hypothesis. The function `laird`, in its improved version, first gives its argument `h` a chance to side-effect by jumping out of the current evaluation. If `h` did not jump, it can then be treated like a value.

Implicit in this reasoning, one can find the assumption that all total morphisms are effect-free. Again, it is refuted by using the current continuation twice: figures 6.21 and 6.22.

The importance of this refutation lies in that it invalidates Laird's claim to have a fully abstract semantics for a language with `callcc` [Lai97]. Hence the situation for full abstraction is not improved by Games models. There still is a tradeoff between models with the full `callcc`, but not fully abstract without the

```
fun laird1 (h :(('a -> '2b) -> void) -> void) =
    fn x =>
    (force' : (('2b -> void) -> void)  -> '2b)
    (fn y => h ((fn z => y (z x))));

    laird1 : ((('a -> '2b) -> void) -> void) -> 'a -> '2b;

fun laird  h =
    (fn u =>
     fn x => force' (fn k =>
                     h (fn f => k (f x))))
    (force' (fn p => h (fn f => p (fn z => z))));

    laird  : ((('a -> '2b) -> void) -> void) -> 'a -> '2b;
```

Figure 6.19: Laird's bootstrap in ML

```
(define (laird h)
  ((lambda (u)
     (lambda (x)
       (call/cc (lambda (k)
                  (h (lambda (f) (k (f x))))))))
   (call/cc (lambda (p)
              (h (lambda (f) (p (lambda (z) z))))))))
```

Figure 6.20: Laird's bootstrap in Scheme

```
val argfct =
    callcc'(fn a =>
            (fn k =>
             invoid (k (fn x =>
                        invoid (k (fn y => x: string)))))
            (force' a));

    argfct : ((string -> string) -> void) -> void;



fun lairddistinguisher testee =
    (fn f =>
       (f "Not Laird.";
        f "Laird."))
    (testee argfct);

lairddistinguisher force';

lairddistinguisher laird;
```

Figure 6.21: Failure of Laird's bootstrap: A distinguishing context in ML

```
(define argfct
  (call/cc (lambda (a)
             ((lambda (k)
                (k (lambda (x) (k (lambda (y) x)))))
              (call/cc a)))))

(define (lairddistinguisher testee)
  ((lambda (f)
     (f "Not Laird.")
     (f "Laird."))
    (testee argfct)))

(lairddistinguisher call/cc)

(lairddistinguisher laird)
```

Figure 6.22: Failure of Laird's bootstrap: A distinguishing context in Scheme

prompt [SF90], and fully abstract models with `callcc` only at base types, as in [KCF92].

A possible formal connection between the refutation of Laird's attempt at bootstrapping `force` and the categorical approach appears to be given by coherence conditions, or more specifically lack thereof. Whereas the unit of the self-adjointness on the right satisfies coherence, the unit for the self-adjointness on the left seems to be inherently *in*decomposable. Intuitively, it seems evident (in the light of the counterexample) that `force` at function type $\sigma \to \tau$ needs to pass the continuation of type $\neg(\sigma \to \tau)$ to its argument and cannot get away with passing something else fabricated from a continuation of type $\neg\tau$. It may be an open problem meriting futher work to state in precisely which algebraic sense `force` is indecomposable (or possibly prime).

6.6 Cross reference to preceding chapters

The programs in this chapter are intended to illustrate a (semantic) point. This relates then to material in other chapters. We give a little link table.

Slogan	ML code	Scheme code	Proposition or remark
`twicecc` not copyable	figure 6.5	figure 6.6	5.2.13
`force` can reify	figure 6.9	figure 6.10	4.4.2
`twicecc` not central	figure 6.12	figure 6.13	implied by 5.2.7

6.7 Discriminating $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ under call by name

We show that the expressive power of `callcc` is sufficient to distinguish the terms $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ under call-by-name. Moreover, the key ingredient for making the distinction is a term that uses its current continuation twice.

Recall the Plotkin call-by-name CPS transform from Definition 3.2.2: $\underline{(\ )}$ extended with `callcc` and `throw`.

$$\begin{array}{rcl}
\underline{x} & = & x\langle k\rangle \\
\underline{\lambda x.M} & = & k\langle f\rangle\{f\langle xk\rangle{=}\underline{M}\} \\
\underline{MN} & = & \underline{M}\{k\langle f\rangle{=}f\langle nk\rangle\{n\langle k\rangle{=}\underline{N}\}\} \\
\underline{\mathtt{callcc}\ M} & = & \underline{M}\{k\langle f\rangle{=}f\langle gk\rangle\{g\langle p\rangle{=}p\langle k\rangle\}\} \\
\underline{\mathtt{throw}\ M\ N} & = & \underline{M}\{k\langle k\rangle{=}\underline{N}\}
\end{array}$$

Here we consider this as an *untyped* transform from untyped λ-calculus (with `callcc`) to untyped CPS calculus.

We define a source language term `argfc` by

$$\texttt{argfc} \stackrel{\text{def}}{=} \texttt{callcc}(\lambda h.\texttt{throw}\, h\, (\lambda x.\texttt{throw}\, h\, (\lambda y.x)))$$

Let $A(k) \stackrel{\text{def}}{=} \underline{\texttt{argfc}}$ be the corresponding CPS term (for current continuation k), that is to say:

$$\begin{array}{ll} & \underline{\texttt{argfc}} \\ = & \underline{\texttt{callcc}(\lambda h.\texttt{throw}\, h\, (\lambda x.\texttt{throw}\, h\, (\lambda y.x)))} \\ = & h\langle k\rangle\{k\langle k\rangle{=}k\langle f\rangle\{f\langle xk\rangle{=}h\langle k\rangle\{k\langle k\rangle{=}k\langle f\rangle\{f\langle yk\rangle{=}x\langle k\rangle\}\}\}\}\{h\langle p\rangle{=}p\langle k\rangle\} \\ = & k\langle f\rangle\{f\langle ap\rangle{=}k\langle f\rangle\{f\langle bq\rangle{=}a\langle q\rangle\}\} \\ \stackrel{\text{def}}{=} & A(k) \end{array}$$

First, we note that $A(k)\{k\langle f\rangle{=}f\langle yk\rangle\} = y\langle k\rangle$.

$$\begin{array}{ll} & A(k)\{k\langle f\rangle{=}f\langle yk\rangle\} \\ = & k\langle f\rangle\{f\langle ap\rangle{=}k\langle f\rangle\{f\langle bq\rangle{=}a\langle q\rangle\}\}\{k\langle f\rangle{=}f\langle yk\rangle\} \\ = & f\langle yk\rangle\{f\langle ap\rangle{=}f\langle yk\rangle\{f\langle bq\rangle{=}a\langle q\rangle\}\} \\ = & y\langle k\rangle \end{array}$$

Now the CPS transform of xx is:

$$\underline{xx} \quad = \quad x\langle k\rangle\{k\langle f\rangle{=}f\langle xk\rangle\}$$

Therefore,

$$\begin{array}{ll} & \underline{(\lambda x.xx)\,\texttt{argfc}} \\ = & k\langle f\rangle\{f\langle xk\rangle{=}\underline{xx}\}\{k\langle f\rangle{=}f\langle xk\rangle\{x\langle k\rangle{=}A(k)\}\} \\ = & \underline{xx}\{x\langle k\rangle{=}A(k)\} \\ = & x\langle k\rangle\{k\langle f\rangle{=}f\langle xk\rangle\}\{x\langle k\rangle{=}A(k)\} \\ = & A(k)\{k\langle f\rangle{=}f\langle xk\rangle\{x\langle k\rangle{=}A(k)\}\} \\ = & A(k)\{k\langle f\rangle{=}f\langle xk\rangle\}\{x\langle k\rangle{=}A(k)\} \\ = & x\langle k\rangle\{x\langle k\rangle{=}A(k)\} \\ = & A(k) \end{array}$$

On the other hand, the CPS transform of $x(\lambda y.xy)$ is:

$$\underline{x(\lambda y.xy)} \quad = \quad x\langle k\rangle\{k\langle f\rangle{=}f\langle nk\rangle\{n\langle k\rangle{=}k\langle g\rangle\{g\langle yk\rangle{=}x\langle k\rangle\{k\langle f\rangle{=}f\langle yk\rangle\}\}\}\}$$

Hence,

$$
\begin{array}{rl}
& \underline{(\lambda x.x(\lambda y.xy))\,\texttt{argfc}} \\
= & k\langle f\rangle\{f\langle xk\rangle{=}\underline{x(\lambda y.xy)}\}\{k\langle f\rangle{=}f\langle xk\rangle\{x\langle k\rangle{=}A(k)\}\} \\
= & \underline{x(\lambda y.xy)}\{x\langle k\rangle{=}A(k)\} \\
= & x\langle k\rangle\{k\langle f\rangle{=}f\langle nk\rangle\{n\langle k\rangle{=}k\langle g\rangle\{g\langle yk\rangle{=}x\langle k\rangle\{k\langle f\rangle{=}f\langle yk\rangle\}\}\}\}\{x\langle k\rangle{=}A(k)\} \\
= & A(k)\{k\langle f\rangle{=}f\langle nk\rangle\{n\langle k\rangle{=}k\langle g\rangle\{g\langle yk\rangle{=}A(k)\{k\langle f\rangle{=}f\langle yk\rangle\}\}\}\} \\
= & A(k)\{k\langle f\rangle{=}f\langle nk\rangle\}\{n\langle k\rangle{=}k\langle g\rangle\{g\langle yk\rangle{=}A(k)\{k\langle f\rangle{=}f\langle yk\rangle\}\}\} \\
= & n\langle k\rangle\{n\langle k\rangle{=}k\langle g\rangle\{g\langle yk\rangle{=}y\langle k\rangle\}\} \\
= & k\langle g\rangle\{g\langle yk\rangle{=}y\langle k\rangle\}
\end{array}
$$

Finally, $A(k)$ and $k\langle g\rangle\{g\langle yk\rangle{=}y\langle k\rangle\}$ can be distinguished.

The terms $\lambda x.xx$ and $\lambda x.(\lambda y.xy)$ are one of the canonical examples cited as evidence of the expressive power of the π-calculus [San95]. While it was originally claimed that this expressive power was due to specific features of the π-calculus, notably nondeterminism, the realisation of the importance of CPS in the translation form λ to π-calculus makes it seem plausible that it is in fact due to the presence of continuations [Davide Sangiorgi, personal communication]. The above can be seen as preliminary evidence of this view. It is perhaps not surprising that we can distinguish $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$, in that the same terms were already used in [Plo75] as a counterexample to show a non-completeness result.

Compare this with Theorem 8.5 and the "conditional η-rule" Corollary 8.4 in [San94]. There Sangiorgi shows that in a Church-Rosser calculus, $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ cannot be distinguished. Roughly, the reason is that in case M diverges, both MM and $M(\lambda y.My)$ diverge; otherwise M and $\lambda y.My$ (y fresh) cannot be distinguished.

This reasoning, valid for a restricted class of calculi, appears to be precisely what gave rise to the flawed assumptions about control operators encountered in the preceding sections.

In the remainder of this section, we formualise the distinction between $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ in Scheme. In order to make Scheme behave like call-by-name, we use a thunking transform, with the thunks being implemented by means of continuations. Concretely, this means that variable occurences need to be replaced by forcings and function arguments need to be thunked. The first is achieved by replacing x with (`call/cc` x) Recall that in Scheme, forcing a CPS thunk is just a special case of `call/cc`. (This is not necessarily the same as Scheme's built in `force`, which may or may not be implemented in this way.) In the two λ-terms and $\lambda x.x(\lambda y.xy)$, the only arguments are values, hence we can thunk them by applying then function `thunk`, and if the argument is a variable that cancels its forcing, i.e.`(thunk(force` x`))`$= x$, so we can just write x.

Notice that the call-by-name semantics of `argfc` is the same as the call-by-value semantics of an almost identical term.

$$\begin{aligned} & \overline{\texttt{callcc}(\lambda h.\texttt{throw}\, h\, (\lambda x.\texttt{throw}\, h\, (\lambda y.\texttt{force}\, x)))} \\ = \; & \underline{\texttt{callcc}(\lambda h.\texttt{throw}\, h\, (\lambda x.\texttt{throw}\, h\, (\lambda y.x)))} \end{aligned}$$

So in Scheme, the distinguishing context will consist essentially of a (call by name) application to $\texttt{callcc}(\lambda h.\texttt{throw}\, h\, (\lambda x.\texttt{throw}\, h\, (\lambda y.\texttt{force}\, x)))$.

For writing the distinguishing context in figure 6.23, we need to take some care in thunking term that are not values. For thunking a value, we can simply apply the function thunk:

```
(define (thunk a)
  (call/cc
   (lambda (k)
     ((call/cc k)
      a))))
```

However, applying `thunk` to a side-effecting term does not succeed in wrapping the side effect into the thunk. In `(thunk (write 'Effect))`, the argument will be evaluated an the effect will not be wrapped into the thunk. That can only be achieved by putting the side-effecting term into the thunking idiom:

```
(define effect-in-thunk
  (call/cc
   (lambda (q)
     ((lambda (p)
        (p (write 'Effect)))
      (call/cc q)))))
```

The printing occurs only when `(call/cc effect-in-thunk)` is forced.

More generally, a computation, as opposed to a value, is thunked by the following idiom:

```
(call/cc
 (lambda (q)
   ((lambda (p)
      (p
```

```
(define lambda-x-xx
  (lambda (x)
    ((call/cc x)
     x)))

(define lambda-x-x-lambda-y-xy
  (lambda (x)
    ((call/cc x)
     (thunk
      (lambda (y)
        ((call/cc x)
         y))))))

(define dist
  (lambda (testee)
    ((lambda (f)
       (f (thunk 1))
       (f (thunk 2)))
     (testee
      (call/cc
       (lambda (q)
         ((lambda (p)
            (p
             (call/cc
              (lambda (h)
                (h
                 (lambda (x)
                   (h
                    (lambda (y)
                      (call/cc x)))))))))
          (call/cc q))))))))
```

Figure 6.23: Distinguishing $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ under call by name

computation to be thunked

```
  ))
(call/cc q))))
```

In figure 6.23, we put all these ingredients, namely, (a variant of) the function `argfc`, the thunking idiom, and a distinguishing context like that in figure 6.15, together to get a distinguishing context for $\lambda x.xx$ and $\lambda x.x(\lambda y.xy)$ under call by name.

`(dist lambda-x-xx)` evaluates to 1, while `(dist lambda-x-x-lambda-y-xy)` evaluates to 2.

Chapter 7

Categorical semantics in $\otimes\neg$-categories

In this chapter, we develop the categorical counterparts of CPS transforms (see chapter 3).

7.1 Call-by-value semantics

Given the notions of λ-abstraction from section 4.5, a simply-typed λ-calculus can be interpreted in a $\otimes\neg$-category.

For call-by-value, control operators are naturally part of such a semantics, as they relate directly to the fundamental operations on the $\neg$ type. Specifically, `callcc` is interpreted as post-composition with the adjoint correspondent

$$[\neg A \to A] = \neg(\neg A \otimes \neg A) \longrightarrow A$$

of the diagonal map $\neg A \longrightarrow \neg A \otimes \neg A$.

7.1.1 Definition (Semantics for call-by-value with `callcc`) Given a $\otimes\neg$-category $\mathcal{K}$, we can give an interpretation $\mathcal{V}[\![-]\!]$ for λ-calculus with control as follows. Types and environments are interpreted as usual, except for the breaking down of arrow types.

$$\begin{aligned}
\mathcal{V}[\![\neg\tau]\!] &\stackrel{\text{def}}{=} \neg\mathcal{V}[\![\tau]\!] \\
\mathcal{V}[\![\sigma \to \tau]\!] &\stackrel{\text{def}}{=} \neg(\mathcal{V}[\![\sigma]\!] \otimes \neg\mathcal{V}[\![\tau]\!]) \\
\mathcal{V}[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] &\stackrel{\text{def}}{=} \mathcal{V}[\![\tau_1]\!] \otimes \cdots \otimes \mathcal{V}[\![\tau_n]\!]
\end{aligned}$$

A judgement $\Gamma \vdash M : \tau$ denotes a morphism $\mathcal{V}[\![\Gamma]\!] \longrightarrow \mathcal{V}[\![\tau]\!]$, defined by induction on M.

$$\mathcal{V}[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash x_j{:}\tau_j]\!] \stackrel{\text{def}}{=} \pi_j$$

$$
\begin{aligned}
\mathcal{V}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] &\stackrel{\text{def}}{=} \overline{\lambda}_{\mathcal{V}[\![\sigma]\!]}\, \mathcal{V}[\![x : \sigma, \Gamma \vdash M : \tau]\!] \\
\mathcal{V}[\![\Gamma \vdash \mathtt{throw}\, M\, N{:}\sigma]\!] &\stackrel{\text{def}}{=} \langle \mathcal{V}[\![\Gamma \vdash N{:}\tau]\!], \mathcal{V}[\![\Gamma \vdash M{:}\neg\tau]\!]\rangle; \mathcal{V}[\![\tau]\!] \otimes \neg\pi_1; \mathsf{apply} \\
\mathcal{V}[\![\Gamma \vdash \mathtt{callcc}\, M : \tau]\!] &\stackrel{\text{def}}{=} \mathcal{V}[\![\Gamma \vdash M{:}\neg\tau \to \tau]\!]; \neg\langle \mathsf{id}_{\neg\mathcal{V}[\![\tau]\!]}, \mathsf{id}_{\neg\mathcal{V}[\![\tau]\!]}\rangle; \mathsf{force} \\
\mathcal{V}[\![\Gamma \vdash MN : \tau]\!] &\stackrel{\text{def}}{=} \langle \mathcal{V}[\![\Gamma \vdash N : \sigma]\!], \mathcal{V}[\![\Gamma \vdash M : \sigma \to \tau]\!]\rangle; \mathsf{apply}
\end{aligned}
$$

What makes the call-by-value semantics work is the fact that all values denote central morphisms, together with the fact that the centre has finite products and that we have central closure.

7.1.1 The naturality of `callcc`

Perhaps the most canonical property of control operators is the naturality of `callcc`, in the sense of the following axiom from [Hof94].

$$V\ (\mathtt{callcc}\ M) \quad = \quad \mathtt{callcc}(\lambda k.V\ (M(\lambda x.k\ (V x))))$$

where V ranges over values, i.e. $V ::= x \mid \lambda x.M$. However, this relies on continuations being a special case of procedures, as in Scheme. With a typing for continuations like that in NJ-SML, instances of this axiom will be ill-typed.

The negation operation suggested by our categorical semantics, definable as $\mathtt{negate} \stackrel{\text{def}}{=} \lambda f.\lambda h.\mathtt{callcc}((\mathtt{throw}\, h) \circ f \circ \mathtt{callcc} \circ \mathtt{throw})$, is useful for adapting this axiom as follows

$$V\ (\mathtt{callcc}\ M) \quad = \quad \mathtt{callcc}(V \circ M \circ (\mathtt{negate}\, V))$$

(For example, let $V =$ `fn n => n + 1` and $M =$ `fn k => throw k 1`. Then V `(callcc` M`)` and `callcc(`V `o` M `o (negate` V`))` both evaluate to 2.) This axiom is sound for our semantics.

7.1.2 Proposition

$$\mathcal{V}[\![\Gamma \vdash V\ (\mathtt{callcc}\ M) : \tau]\!] = \mathcal{V}[\![\Gamma \vdash \mathtt{callcc}(V \circ M \circ (\mathtt{negate}\, V)) : \tau]\!]$$

7.2 Plotkin call-by-name semantics and variants

Plotkin call-by-name is a slight variation on call-by-name obtained by the technique of thunking arguments in applications and forcing variables; this is a categorical analogue of [HD95].

We consider the doubly-negated $\neg\neg A$ as the type of lazy data of type A.

7.2.1 Definition (Plotkin Call-by-name semantics) Given a $\otimes\neg$-category, we define the Plotkin call-by-name semantics $\mathcal{L}[\![-]\!]$ as follows.

$$\begin{aligned}
\mathcal{L}[\![\sigma \to \tau]\!] &\stackrel{\text{def}}{=} \neg(\neg\neg\mathcal{P}[\![\sigma]\!] \otimes \neg\mathcal{P}[\![\tau]\!]) \\
\mathcal{L}[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] &\stackrel{\text{def}}{=} \neg\neg\mathcal{P}[\![\tau_1]\!] \otimes \cdots \otimes \neg\neg\mathcal{P}[\![\tau_n]\!]
\end{aligned}$$

Again, a judgement $\Gamma \vdash M : \tau$ denotes a morphism $\mathcal{P}[\![\Gamma]\!] \longrightarrow \mathcal{P}[\![\tau]\!]$.

$$\begin{aligned}
\mathcal{P}[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash x_j{:}\tau_j]\!] &\stackrel{\text{def}}{=} \pi_j; \mathsf{force} \\
\mathcal{P}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] &\stackrel{\text{def}}{=} \overline{\lambda}_{\mathcal{P}[\![\sigma]\!]}\mathcal{P}[\![x : \sigma, \Gamma \vdash M : \tau]\!] \\
\mathcal{P}[\![\Gamma \vdash MN : \tau]\!] &\stackrel{\text{def}}{=} \langle \overline{\lambda}_{\mathbf{1}}\, \mathcal{P}[\![\Gamma \vdash N : \sigma]\!], \mathcal{P}[\![\Gamma \vdash M : \sigma \to \tau]\!]\rangle; \mathsf{apply}
\end{aligned}$$

The Plotkin call-by-name semantics satisfies the full β law by thunking arguments at the point of application. Thus arguments are always central. However, the application map is still the same as in call-by-value; the "jump with arguments", apply, which is not central.

The Plotkin semantics rests on the thunking corollary 4.5.3 along with the central closure 4.5.2.

7.2.2 Remark We now have the categorical framework in place in order to talk more abstractly about some of the issues addressed in terms of name-passing in chapter 3. We mentioned in remark 3.1.5 two possible choices for a semantics with the call-by-name typing. These correspond to the two passages in the naturality square for thunk. Because the naturality does not hold in general (only in a subcategory), there really is a choice.

As explained in [DH94], the Reynolds and modified Reynolds have the same semantics of function types, but they differ in the choice of when the delayed argument is forced.

7.2.3 Definition Given a $\otimes\neg$-category, we define the Reynolds call-by-value semantics $\mathcal{R}[\![-]\!]$ as follows.

$$\begin{aligned}
\mathcal{R}[\![\sigma \to \tau]\!] &\stackrel{\text{def}}{=} \neg(\neg\neg\mathcal{R}[\![\sigma]\!] \otimes \neg\mathcal{R}[\![\tau]\!]) \\
\mathcal{R}[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] &\stackrel{\text{def}}{=} \neg\neg\mathcal{R}[\![\tau_1]\!] \otimes \cdots \otimes \neg\neg\mathcal{R}[\![\tau_n]\!]
\end{aligned}$$

A judgement $\Gamma \vdash M : \tau$ denotes a morphism $\mathcal{R}[\![\Gamma]\!] \longrightarrow \mathcal{R}[\![\tau]\!]$.

$$\begin{aligned}
\mathcal{R}[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash x_j{:}\tau_j]\!] &\stackrel{\text{def}}{=} \pi_j; \mathsf{force} \\
\mathcal{R}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] &\stackrel{\text{def}}{=} \overline{\lambda}_{\mathcal{R}[\![\sigma]\!]}((\mathsf{force}; \mathsf{thunk}) \otimes \mathcal{R}[\![\Gamma]\!]; \mathcal{R}[\![x : \sigma, \Gamma \vdash M : \tau]\!]) \\
\mathcal{R}[\![\Gamma \vdash MN : \tau]\!] &\stackrel{\text{def}}{=} \langle \overline{\lambda}_{\mathbf{1}}\, \mathcal{R}[\![\Gamma \vdash N : \sigma]\!], \mathcal{R}[\![\Gamma \vdash M : \sigma \to \tau]\!]\rangle; \mathsf{apply}
\end{aligned}$$

7.2.4 Definition Given a $\otimes\neg$-category, we define the modified Reynolds call-by-value semantics $\mathcal{R}'[\![-]\!]$ as follows.

$$\begin{array}{rcl} \mathcal{R}'[\![\sigma \to \tau]\!] & \stackrel{\text{def}}{=} & \neg(\neg\neg\mathcal{R}'[\![\sigma]\!] \otimes \neg\mathcal{R}'[\![\tau]\!]) \\ \mathcal{R}'[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] & \stackrel{\text{def}}{=} & \mathcal{R}'[\![\tau_1]\!] \otimes \cdots \otimes \mathcal{R}'[\![\tau_n]\!] \end{array}$$

A judgement $\Gamma \vdash M : \tau$ denotes a morphism $\mathcal{R}'[\![\Gamma]\!] \longrightarrow \mathcal{R}'[\![\tau]\!]$.

$$\begin{array}{rcl} \mathcal{R}'[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash x_j{:}\tau_j]\!] & \stackrel{\text{def}}{=} & \pi_j \\ \mathcal{R}'[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] & \stackrel{\text{def}}{=} & \overline{\lambda}_{\mathcal{R}'[\![\sigma]\!]}(\mathsf{force} \otimes \mathcal{R}'[\![\Gamma]\!]; \mathcal{R}'[\![x : \sigma, \Gamma \vdash M : \tau]\!]) \\ \mathcal{R}'[\![\Gamma \vdash MN : \tau]\!] & \stackrel{\text{def}}{=} & \langle \overline{\lambda}_{\mathbf{1}}\, \mathcal{R}'[\![\Gamma \vdash N : \sigma]\!], \mathcal{R}'[\![\Gamma \vdash M : \sigma \to \tau]\!]\rangle; \mathsf{apply} \end{array}$$

7.3 Uncurrying call-by-name semantics

The uncurrying call-by-name semantics relies on the variant λ-abstraction $\underline{\underline{\lambda}}$ and application $\underline{\underline{\mathsf{apply}}}$.

7.3.1 Definition (Semantics for uncurrying call-by-name)

$$\mathcal{N}[\![\tau_1 \to \ldots \to \tau_n \to b]\!] \stackrel{\text{def}}{=} \neg(\mathcal{N}[\![\tau_1]\!] \otimes \cdots \otimes \mathcal{N}[\![\tau_n]\!] \otimes \neg\mathcal{N}[\![b]\!])$$

$$\begin{array}{rcl} \mathcal{N}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_j : \tau]\!] & \stackrel{\text{def}}{=} & \pi_j \\ \mathcal{N}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] & \stackrel{\text{def}}{=} & \underline{\underline{\lambda}}_{\mathcal{N}[\![\sigma]\!]}\, \mathcal{N}[\![x : \sigma, \Gamma \vdash M : \tau]\!] \\ \mathcal{N}[\![\Gamma \vdash MN : \tau]\!] & \stackrel{\text{def}}{=} & \langle \mathcal{N}[\![\Gamma \vdash N : \sigma]\!], \mathcal{N}[\![\Gamma \vdash M : \sigma \to \tau]\!]\rangle; \underline{\underline{\mathsf{apply}}} \end{array}$$

The uncurrying call-by-name semantics, by contrast with the Plotkin one, uses an application map that is itself central (see 4.5.5). Hence all denotations are central. This it what make this semantics validate both the full β and η.

We can now explain more conceptually the invariant (lemma 3.3.1) that made the syntactic soundness proof of the uncurrying call-by-name CPS transform work: it is just one of the many equivalent characterisation of centrality, namely thunkability.

Both the denotations of variables and the application map used by the uncurrying semantics are central (denotations of λ-abstraction are central for all semantics considered here). This is what makes the uncurrying semantics genuinely call-by-name. For the Plotkin call-by-name semantics, by contrast, not even the denotations of variables are central.

7.3.2 Proposition For the special case of the term model, that is $\mathcal{K} = \mathcal{K}(\mathrm{CPS})$, the denotation of a judgement is the equivalence class of its CPS transform.

$$\begin{array}{rcl} \mathcal{V}[\![\Gamma \vdash M : \tau]\!] & = & [(\!|\Gamma|\!), k : \neg(\!|\tau|\!) \vdash (\!|M|\!)(k)] \\ \mathcal{P}[\![\Gamma \vdash M : \tau]\!] & = & [\mathcal{L}(\!|\Gamma|\!), k : \neg(\!|\tau|\!) \vdash (\!|M|\!)(k)] \\ \mathcal{R}[\![\Gamma \vdash M : \tau]\!] & = & [\mathcal{R}(\!|\Gamma|\!), k : \neg(\!|\tau|\!) \vdash (\!|M|\!)(k)] \\ \mathcal{R}'[\![\Gamma \vdash M : \tau]\!] & = & [\mathcal{R}'(\!|\Gamma|\!), k : \neg(\!|\tau|\!) \vdash (\!|M|\!)(k)] \\ \mathcal{N}[\![\Gamma \vdash M : \tau]\!] & = & [\mathcal{N}(\!|\Gamma|\!), k : \neg\mathcal{N}(\!|\tau|\!) \vdash \mathcal{N}(\!|M|\!)(k)] \end{array}$$

For the shorthand transformation this reads as follows

$$\begin{array}{c} \mathcal{V}[\![\Gamma \vdash M : \tau]\!] = [\overline{\Gamma}, k : \neg\tau \vdash \overline{M}] \\ \mathcal{P}[\![\Gamma \vdash M : \tau]\!] = [\underline{\Gamma}, k : \neg\tau \vdash \underline{M}] \\ \mathcal{N}[\![\Gamma \vdash M : \tau]\!] = [\underline{\underline{\Gamma}}, k : \neg\tau \vdash \underline{\underline{M}}] \end{array}$$

7.4 State and meta-continuation-passing

A very elementary construction on a premomoidal category is to "add state": each morphism takes and returns an additional argument. This can be regarded as depending on, and possibly modifying, some global variable. Whiel this is not a very satisfactory account of state in programming languages, it is sufficient to give a categorical counterpart of meta-continuation passing style 3.2.4 on page 50. The combination of first-class control and (even a single piece of global) state seems to be a rather powerful one; see [Fil94b].

$$\begin{array}{rcl} \mathbf{Ob}\mathcal{K}!S & = & \mathbf{Ob}\mathcal{K} \\ \mathcal{K}!S(A,B) & = & \mathcal{K}(S \otimes A, S \otimes B) \\ \neg^S A & = & \neg(S \otimes A) \\ \neg^S f & = & S \otimes \neg f \\ \phi^S_A f & = & \phi_{S\otimes A} f \\ \mathsf{thunk}^S & = & S \otimes \mathsf{pair} \end{array}$$

7.4.1 Remark In $\mathcal{K}!S$, we have operations for dereferencing and assigning to the single piece of state:

$$\begin{array}{rcl} := & : & S \longrightarrow \mathbf{1} \\ ! & : & ! \longrightarrow S \end{array}$$

These are just the second projection

$$:= \stackrel{\text{def}}{=} \pi_2 : S \otimes S \longrightarrow S = \mathbf{1} \otimes S$$

the diagonal map

$$! \stackrel{\text{def}}{=} \delta_S : S \otimes \mathbf{1} = S \longrightarrow S \otimes S$$

in the base category $\mathcal{K}$. Note that, in the CPS term model $! = [sk \vdash k\langle ss\rangle]$ and $:= = [sxk \vdash k\langle x\rangle]$. In that sense, `!` is essentially the same as `callcc`, and `:=` the same as `throw`. There may be a connection with the relationship, mentioned in the introduction, between figures 1.7 and 1.8, and figures 1.5 and 1.6. Compare also figures 6.16 and 6.15 in Chapter 6.

7.4.2 Conjecture We conjecture that, under some "mild" conditions on $\mathcal{K}$, if $\mathcal{K}$ is a $\otimes\neg$-category and $S \in \mathbf{Ob}\mathcal{K}$, then $\mathcal{K}!S$ is a $\otimes\neg$-category.

Meta-continuation-passing style arises as the special case $S = \neg R$. Expressions of type R can be aborted and control-delimited.

7.5 Categorical semantics for CPS calculus

Assigning a morphism to each CPS calculus judgement is made somewhat awkward by the fact that these judgements have only premises (typing for names), but no conclusions (type for the whole term), so that there does not seem to be a canonical choice for the codomain of its denotation. This is perhaps surprising, but we would argue that CPS is so low-level that even composition is not fundamental, but a relatively involved idiom from its perspective.

We could either choose a "dummy" codomain (the corresponding continuation never being used) or single out one of the names in a judgement as the current continuation corresponding to the codomain.

Compared to the (syntactic) CPS transforms from the CPS literature, the first of these is analogous to a Continuation Grabbing Style [Sab96] transformation, while the second would amount to a Back to Direct Style [Dan94, DL92] transform.

We sketch each of those, before giving a more detailed account of a Direct Style semantics for a small fragment of CPS calculus.

7.5.1 Continuation Grabbing Style semantics for CPS-calculus

A Continuation Grabbing Style semantics for CPS-calculus could be defined as follows.

$$[\![\Gamma \vdash M]\!] : [\![\Gamma]\!] \longrightarrow \neg \mathbf{1}$$

$$\begin{array}{rl} & [\![\Gamma, \vec{x} : \vec{\sigma}, k : \neg\vec{\sigma} \vdash k\langle\vec{x}\rangle]\!] \\ = & \pi_j \end{array}$$

$$\begin{array}{rl} & [\![\Gamma \vdash M\{n\langle\vec{x}\rangle{=}N\}]\!] \\ = & \delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma]\!] \otimes \mathsf{thunk}_{\mathbf{1}}; [\![\Gamma]\!] \otimes \left(\phi_{[\![\Gamma]\!]}[\![\Gamma, n : \neg\vec{\sigma} \vdash M]\!]\right); [\![\Gamma, \vec{x} : \vec{\vec{\sigma}} \vdash N]\!] \end{array}$$

7.5.2 Back to Direct Style semantics for CPS-calculus

A Back to Direct Style would have to solve the problem of how to choose a current continuation, corresponding to the codomain of the denotation, in each CPS calculus judgement.

One could introduce conventions for singling out one variable (the last, say) in a judgement as the current continuation. (Incidentally, it is not trivial that this is always possible; we conjecture that in the non-recursive CPS calculus, for each derivable judgement at least one variable in the typing assumptions is assigned a continuation type.)

A judgement $\Gamma, k : \neg\tau \vdash M$ would then denote a morphism

$$[\![\Gamma]\!] \longrightarrow [\![\tau]\!]$$

The problem is how to define this inductively on the structure of M.

If, in a binding $M\{n\langle\vec{x}\rangle{=}N\}$, k is free in N, but not in M, then the denotation of $\Gamma, k : \neg\tau \vdash M\{n\langle\vec{x}\rangle{=}N\}$ should be a straightforward composition [1] of the denotations $[\![\Gamma, n : \neg\vec{\sigma} \vdash M]\!]$ and $[\![\Gamma, \vec{x} : \vec{\sigma} \vdash N]\!]$.

But in general k may occur anywhere in M or N or both, requiring a complicated case analysis. (In [Dan94, DL92], this necessitates intricate techniques for digging out a current continuation.)

We restrict ourselves to a fragment of CPS so simple that a few cases are enough: that is we consider only the linear unary CPS calculus. This is a fairly severe restriction: without polyadic continuations, we loose the ability to translate functions. On the other hand this simple fragment has a clean categorical semantics: it is the internal language of a self-adjointness.

It is straightforward to build a category equipped with a self-adjoint functor from the syntax of simply-typed, linear unary CPS calculus along the lines of definition 5.2.1. We focus on the other direction.

[1] In the co-Kleisli category $\mathcal{K}_{[\![\Gamma]\!]}$

7.5.1 Definition Given a category $\mathcal{K}$ together with a self-adjoint functor $\neg : \mathcal{K}^{\mathrm{op}} \longrightarrow \mathcal{K}$ with unit force, we define a semantics for the linear unary CPS calculus.

$$\begin{array}{rcll}
[\![xk \vdash k\langle x\rangle]\!] & = & \mathsf{id} & \\
[\![xk \vdash x\langle k\rangle]\!] & = & \mathsf{force} & \\
[\![xk \vdash M\{n\langle y\rangle{=}N\}]\!] & = & [\![xn \vdash M]\!];[\![yk \vdash N]\!] & , k \in \mathsf{FV}(N) \\
[\![xk \vdash M\{n\langle y\rangle{=}N\}]\!] & = & \neg[\![yx \vdash N]\!];[\![nk \vdash M]\!] & , k \in \mathsf{FV}(M)
\end{array}$$

7.5.2 Lemma $[\![xk \vdash M]\!] = [\![yh \vdash M[xk \mapsto yh]]\!]$

7.5.3 Lemma The denotation of a judgement $kx \vdash M$ with the variables swapped is the adjoint correspondent of the denotation of the original judgement $xk \vdash M$.

$$\neg[\![xk \vdash M]\!];\mathsf{force} = [\![kx \vdash M]\!]$$

Proof By induction on M.

$k\langle x\rangle$

$$\begin{array}{rl}
 & \neg[\![xk \vdash k\langle x\rangle]\!];\mathsf{force} \\
= & \neg\mathsf{id};\mathsf{force} \\
= & \mathsf{force} \\
= & [\![xk \vdash x\langle k\rangle]\!]
\end{array}$$

$x\langle k\rangle$

$$\begin{array}{rl}
 & \neg[\![xk \vdash x\langle k\rangle]\!];\mathsf{force} \\
= & \neg\mathsf{force};\mathsf{force} \\
= & \mathsf{id} \\
= & [\![xk \vdash k\langle x\rangle]\!]
\end{array}$$

$M\{n\langle x\rangle{=}N\}$ First, let $k \in \mathsf{FV}(N)$.

$$\begin{array}{rl}
 & \neg[\![xk \vdash M\{n\langle y\rangle{=}n\}]\!];\mathsf{force} \\
= & \neg([\![xn \vdash M]\!];[\![yk \vdash N]\!]);\mathsf{force} \\
= & \neg[\![yk \vdash N]\!];\neg[\![xn \vdash M]\!];\mathsf{force} \\
= & \neg[\![yk \vdash N]\!];[\![nx \vdash M]\!] \\
= & [\![kx \vdash M\{n\langle y\rangle{=}N\}]\!]
\end{array}$$

Because of the naturality of and the triangular identity for force, we have:

$$\begin{aligned} & \neg(\neg[\![xk \vdash M]\!];\mathsf{force});\mathsf{force} \\ = {} & \neg\mathsf{force};\neg\neg[\![xk \vdash M]\!];\mathsf{force} \\ = {} & \neg\mathsf{force};\mathsf{force};[\![xk \vdash M]\!] \\ = {} & [\![xk \vdash M]\!] \end{aligned}$$

□

7.5.4 Proposition The semantics $[\![\_]\!]$ of linear unary CPS-calculus in definition 7.5.1 is sound with respect to the axioms of the calculus given in definition 2.4.1.

Proof We show for each axiom $M_1 = M_2$ that $[\![M_1]\!] = [\![M_2]\!]$.

Jmp

$$\begin{aligned} & [\![n\langle x\rangle\{n\langle y\rangle{=}N\}]\!] \\ = {} & [\![xn \vdash n\langle x\rangle]\!];[\![yk \vdash N]\!] \\ = {} & \mathsf{id};[\![yk \vdash N]\!] \\ = {} & [\![yk \vdash N]\!] \\ = {} & [\![xk \vdash M[y \mapsto x]]\!] \end{aligned}$$

$$\begin{aligned} & [\![xk \vdash n\langle k\rangle\{n\langle y\rangle{=}N\}]\!] \\ = {} & \neg[\![yx \vdash N]\!];[\![nk \vdash n\langle k\rangle]\!] \\ = {} & \neg[\![yx \vdash N]\!];\mathsf{force} \\ = {} & [\![xy \vdash N]\!] \\ = {} & [\![xk \vdash N[y \mapsto k]]\!] \end{aligned}$$

Eta

$$\begin{aligned} & [\![xk \vdash M\{n\langle y\rangle{=}k\langle y\rangle\}]\!] \\ = {} & [\![xn \vdash M]\!];[\![yk \vdash k\langle y\rangle]\!] \\ = {} & [\![xn \vdash M]\!];\mathsf{id} \\ = {} & [\![xn \vdash M]\!] \\ = {} & [\![xk \vdash M[n \mapsto k]]\!] \end{aligned}$$

$$
\begin{aligned}
& [\![xk \vdash M\{n\langle y\rangle{=}x\langle y\rangle\}]\!] \\
= \quad & \neg[\![yx \vdash x\langle y\rangle]\!]; [\![nk \vdash M]\!] \\
= \quad & \mathsf{id}; [\![nk \vdash M]\!] \\
= \quad & [\![nk \vdash M]\!] \\
= \quad & [\![xk \vdash M[n \mapsto x]]\!]
\end{aligned}
$$

Float-L: $n \in \mathsf{FV}(L)$

- $k \in \mathsf{FV}(M)$

$$
\begin{aligned}
& [\![xk \vdash L\{m\langle y\rangle{=}M\}\{n\langle z\rangle{=}N\}]\!] \\
= \quad & \neg[\![zx \vdash N]\!]; [\![nk \vdash L\{m\langle y\rangle{=}M\}]\!] \\
= \quad & \neg[\![zx \vdash N]\!]; [\![nm \vdash L]\!]; [\![yk \vdash M]\!] \\
= \quad & [\![xm \vdash L\{n\langle z\rangle{=}N\}]\!]; [\![yk \vdash M]\!] \\
= \quad & [\![xk \vdash L\{n\langle z\rangle{=}N\}\{m\langle y\rangle{=}M\}]\!]
\end{aligned}
$$

- $k \in \mathsf{FV}(N)$

$$
\begin{aligned}
& [\![xk \vdash L\{m\langle y\rangle{=}M\}\{n\langle z\rangle{=}N\}]\!] \\
= \quad & \neg[\![yx \vdash M]\!]; [\![mn \vdash L]\!]; [\![zk \vdash N]\!] \\
= \quad & \neg[\![yx \vdash M]\!]; [\![mn \vdash L\{n\langle z\rangle{=}N\}]\!] \\
= \quad & [\![xk \vdash L\{n\langle z\rangle{=}N\}\{m\langle y\rangle{=}M\}]\!]
\end{aligned}
$$

Float-R: $n \in \mathsf{FV}(M)$

- $k \in \mathsf{FV}(N)$

$$
\begin{aligned}
& [\![xk \vdash L\{m\langle y\rangle{=}M\}\{n\langle z\rangle{=}N\}]\!] \\
= \quad & ([\![xm \vdash L]\!]; [\![yn \vdash M]\!];)[\![zk \vdash N]\!] \\
= \quad & [\![xm \vdash L]\!]; ([\![yn \vdash M]\!]; [\![yk \vdash N]\!]) \\
= \quad & [\![xm \vdash L]\!]; [\![yk \vdash M\{n\langle y\rangle{=}N\}]\!] \\
= \quad & [\![xk \vdash L\{m\langle y\rangle{=}M\{n\langle z\rangle{=}N\}\}]\!]
\end{aligned}
$$

- $k \in \mathsf{FV}(L)$

$$
\begin{aligned}
& [\![xk \vdash L\{m\langle y\rangle{=}M\}\{n\langle z\rangle{=}N\}]\!] \\
= \quad & \neg[\![zx \vdash N]\!]; [\![nk \vdash L\{m\langle y\rangle{=}M\}]\!] \\
= \quad & \neg[\![zx \vdash N]\!]; \neg[\![yn \vdash M]\!]; [\![mk \vdash L]\!]
\end{aligned}
$$

$$
\begin{aligned}
&= \neg([\![yn \vdash M]\!];[\![zx \vdash N]\!]);[\![mk \vdash L]\!] \\
&= \neg[\![yx \vdash M\{n\langle z\rangle{=}N\}]\!];[\![mk \vdash L]\!] \\
&= [\![xk \vdash L\{m\langle y\rangle{=}M\{n\langle z\rangle{=}N\}\}]\!]
\end{aligned}
$$

□

We have established that an extremely distilled version of CPS, a unary name-passing calculus, is the internal language of a self-adjointness. From the point of view of categorical semantics, an intriguing question would be what, if anything, in the categorical structure makes its internal language a name-passing calculus.

Chapter 8

Indexed ¬-categories

In Chapter 4, we have mentioned that we can see the continuation functor as indexed. Here we develop this point of view. Although relevant for continuation semantics, the possibility to use either indexed categories or premonoidal structure for the semantics of environments arises in denotational semantics in general, so we treat it at its natural level of generality.

This chapter presents joint work with John Power [PT97]; in particular, the result (8.3.4 below) on which the connection is built is due to him.

8.1 Environments as indices

Traditionally in denotational semantics, there have been two categorical ways of modelling environments. The first is given by finite products in a Cartesian closed category, as for instance in modelling the simply typed λ-calculus. Over the years, that has gradually been extended. For instance, in order to model partiality, one must generalise from finite product structure to symmetric monoidal structure; and more recently, that has been further generalised to the notion of symmetric premonoidal structure [PR97].

A premonoidal category is essentially a monoidal category except that the tensor need only be a functor in two variables separately, and not necessarily a bifunctor: given maps $f : A \to A'$ and $g : B \to B'$, the evident two maps from $A \otimes B$ to $A' \otimes B'$ may differ. Such structures arise naturally in the presence of computational effects, where the difference between these two maps is a result of sensitivity to evaluation order. So that is the structure we need in order to model environments in the presence of continuations or other such strong computational effects. A program phrase in environment Γ is modelled by a morphism in the premonoidal category with domain $[\![\Gamma]\!]$.

The second approach to modelling environments categorically, also used to model the simply typed λ-calculus, is based on indexed categories with structure, and has been heavily advocated, although not introduced, by Bart Jacobs [Jac92]: the slogan is that contexts, which we call environments, are indices for the categories in which the terms definable in that context are modelled. Here, a program phrase in environment Γ is modelled by an element $\mathbf{1} \longrightarrow [\![\tau]\!]$ in a category that implicitly depends on Γ, i.e., by an arrow from $\mathbf{1}$ to $[\![\tau]\!]$ in the fibre of the indexed category over $[\![\Gamma]\!]$. We consider a weak version of indexed category with structure, called a κ-category, implicit in recent work by Masahito Hasegawa [Has95]. In the setting of indexed categories, various binding constructs can be studied. A κ-category has a weak first order notion of binding, given by the assertion that reindexing along projections has a left adjoint. In programming terms, that corresponds to a special form that binds an identifier but is not reifying in the sense that it does not produce a *first class* function. Hasegawa [Has95] compares it to `lambda` in *early* LISP.

The first major result of this chapter is to prove the above two models of environments equivalent. More precisely, we show that every symmetric premonoidal category with a little more of the structure cited above, gives rise to a κ-category, and that this gives a bijection between the classes of symmetric premonoidal categories with such structure and κ-categories. The extra structure we need on a symmetric premonoidal category $\mathcal{K}$ is a category with finite products $\mathcal{C}$ and an identity on objects strict symmetric premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$. At first sight, that may seem a somewhat complex structure, but in fact, as made precise in [Pow], it is particularly natural category theoretic structure, more so than that of premonoidal structure alone, as it is algebraic structure.

Related Work

The relationship between symmetric premonoidal categories and κ-categories is related to work by Blute, Cockett, and Seely [RBS]. Implicit in their work is the construction which, to a symmetric premonoidal category with a little added structure, assigns a κ-category. The latter are closely related to their context categories. Identifying precisely which indexed categories thus arise did not appear in their work.

Bart Jacobs' thesis [Jac91] championed the view of contexts as "*indices* for the terms and types derivable in that context." We believe this to be relevant not only to type theory but also to the modelling of environments in computer science, and we use it for that purpose in our third approach to continuation semantics.

Ong [Ong96] also uses a fibration to model environments for his categorical formulation of the $\lambda\mu$-calculus [Par92]. As this calculus is an extension of the call-by-name λ-calculus, Ong can assume every fibre to be Cartesian closed. However, for call-by-value programming languages like ML or Scheme, one cannot assume Cartesian closure. (And even if one were to assume call-by-name, the intended meaning of `callcc` would be less than clear.)

8.2 Premonoidal categories

In this section, we recall the definitions of premonoidal category and strict premonoidal functor, and symmetries for them, as introduced in [PR97] and further studied in [Pow]. We also develop a basic construction on a premonoidal category that we will need later. A premonoidal category is a generalisation of the concept of monoidal category: it is essentially a monoidal category except that the tensor need only be a functor of two variables and not necessarily be bifunctorial, i.e., given maps $f : A \longrightarrow B$ and $f' : A' \longrightarrow B'$, the evident two maps from $A \otimes A'$ to $B \otimes B'$ may differ.

Historically, for instance for the simply typed λ-calculus, environments have been modelled by finite products. More recently, monoidal structure has sometimes been used, for instance when one wants to incorporate an account of partiality [RR88]. In the presence of stronger computational effects, an even weaker notion is required. If the computational effects are strong enough for the order of evaluation of $f : A \longrightarrow B$ and $f' : A' \longrightarrow B'$ to be observable, as for instance in the case of continuations, then the monoidal laws cannot be satisfied. The leading example for us of such stronger computational effects are those given by continuations. However, for a simple example of a premonoidal category that may be used for a crude account of state [PR97], consider the following.

8.2.1 Example Given a symmetric monoidal category $\mathcal{C}$ together with a specified object S, define the category $\mathcal{K}$ to have the same objects as $\mathcal{C}$, with $\mathcal{K}(A, B) = \mathcal{C}(S \otimes A, S \otimes B)$, and with composition in $\mathcal{K}$ determined by that of $\mathcal{C}$. For any object A of $\mathcal{C}$, one has functors $A \otimes - : \mathcal{K} \longrightarrow \mathcal{K}$ and $- \otimes A : \mathcal{K} \longrightarrow \mathcal{K}$, but they do not satisfy the bifunctoriality condition above, hence do not yield a monoidal structure on $\mathcal{K}$. They do yield a premonoidal structure, as we define below.

In order to make precise the notion of a premonoidal category, we need some auxiliary definitions.

8.2.2 Definition A *binoidal category* is a category $\mathcal{K}$ together with, for each object A of $\mathcal{K}$, functors $h_A : \mathcal{K} \longrightarrow \mathcal{K}$ and $k_A : \mathcal{K} \longrightarrow \mathcal{K}$ such that for each pair (A, B) of objects of $\mathcal{K}$, $h_A B = k_B A$. The joint value is denoted $A \otimes B$.

8.2.3 Definition An arrow $f : A \longrightarrow A'$ in a binoidal category is *central* if for every arrow $g : B \longrightarrow B'$, the following diagrams commute:

$$\begin{tikzcd} A \otimes B \arrow[r, "A\otimes g"] \arrow[d, "f\otimes B"'] & A \otimes B' \arrow[d, "f\otimes B'"] \\ A' \otimes B \arrow[r, "A'\otimes g"] & A' \otimes B' \end{tikzcd} \qquad \begin{tikzcd} B \otimes A \arrow[r, "g\otimes A"] \arrow[d, "B\otimes f"'] & B' \otimes A \arrow[d, "B'\otimes f"] \\ B \otimes A' \arrow[r, "g\otimes A'"] & B' \otimes A' \end{tikzcd}$$

Moreover, given a binoidal category $\mathcal{K}$, a natural transformation $\alpha : g \Longrightarrow h : \mathcal{B} \longrightarrow \mathcal{K}$ is called *central* if every component of α is central.

8.2.4 Definition A *premonoidal category* is a binoidal category $\mathcal{K}$ together with an object I of $\mathcal{K}$, and central natural isomorphisms a with components $(A \otimes B) \otimes C \longrightarrow A \otimes (B \otimes C)$, l with components $A \longrightarrow A \otimes I$, and r with components $A \longrightarrow I \otimes A$, subject to two equations: the pentagon expressing coherence of a, and the triangle expressing coherence of l and r with respect to a.

Now we have the definition of a premonoidal category, it is routine to verify that Example 8.2.1 is an example of one. There is a general construction that yields premonoidal categories too: given a strong monad T on a symmetric monoidal category $\mathcal{C}$, the Kleisli category **Kleisli**(T) for T is always a premoidal category, with the functor from $\mathcal{C}$ to **Kleisli**(T) preserving premonoidal structure strictly: of course, a monoidal category such as $\mathcal{C}$ is trivially a premonoidal category. That construction is fundamental, albeit implicit, in Eugenio Moggi's work on monads as notions of computation [Mog89], as explained in [PR97].

8.2.5 Definition Given a premonoidal category $\mathcal{K}$, define the *centre* of $\mathcal{K}$, denoted $\mathsf{Z}(\mathcal{K})$, to be the subcategory of $\mathcal{K}$ consisting of all the objects of $\mathcal{K}$ and the central morphisms.

For an example of the centre of a premonoidal category, consider Example 8.2.1 for the case of $\mathcal{C}$ being the category Set of small sets, with symmetric monoidal structure given by finite products. Suppose S has at least two elements. Then the centre of $\mathcal{K}$ is precisely Set. In general, given a strong monad on a symmetric monoidal category, the base category $\mathcal{C}$ need not be the centre of **Kleisli**(T), but, modulo a faithfulness condition sometimes called the mono requirement [Mog89, PR97], must be a subcategory of the centre.

The functors h_A and k_A preserve central maps. So we have

8.2.6 Proposition The centre of a premonoidal category is a monoidal category.

This proposition allows us to prove a coherence result for premonoidal categories, directly generalising the usual coherence result for monoidal categories. Details appear in [PR97].

8.2.7 Definition A *symmetry* for a premonoidal category is a central natural isomorphism with components $c : A \otimes B \longrightarrow B \otimes A$, satisfying the two conditions $c^2 = 1$ and equality of the evident two maps from $(A \otimes B) \otimes C$ to $C \otimes (A \otimes B)$. A *symmetric* premonoidal category is a premonoidal category together with a symmetry.

All of the examples of premonoidal categories we have discussed so far are symmetric, and in fact, symmetric premonoidal categories are those of primary interest to us, and seem to be those of primary interest in denotational semantics in general. For an example of a premonoidal category that is not symmetric, consider, given any category $\mathcal{C}$, the category $End_u(\mathcal{C})$ whose objects are functors from $\mathcal{C}$ to itself, and for which an arrow from h to k is a $\mathcal{C}$-indexed family of arrows $\alpha(A) : h(A) \longrightarrow k(A)$ in $\mathcal{C}$, i.e., what would be a natural transformation from h to k but without assuming commutativity of the naturality squares. Then, this category, together with the usual composition of functors, has the structure of a strict premonoidal category, i.e., a premonoidal category in which all the structural isomorphisms are identities, which is certainly not symmetric.

8.2.8 Definition A *strict premonoidal functor* is a functor that preserves all the structure and sends central maps to central maps.

One may similarly generalise the definition of strict symmetric monoidal functor to strict symmetric premonoidal functor.

In order to compare the various models of environments in the next section, we need to study a construction that, to a premonoidal category, assigns a $\mathcal{C}at$-valued functor.

8.2.9 Definition A *comonoid* in a premonoidal category $\mathcal{K}$ consists of an object $\mathbf{C}$ of $\mathcal{K}$, and central maps $\delta : \mathbf{C} \longrightarrow \mathbf{C} \otimes \mathbf{C}$ and $\nu : \mathbf{C} \longrightarrow I$ making the usual associativity and unit diagrams commute.

It follows from centrality of the two maps in the definition of comonoid that one has the usual coherence for a comonoid, i.e., n-fold associativity is well defined, and comultiple products with counits are also well defined.

8.2.10 Definition A *comonoid map* from $\mathbf{C}$ to $\mathbf{D}$ in a premonoidal category $\mathcal{K}$ is a central map $f : \mathbf{C} \longrightarrow \mathbf{D}$ that commutes with the comultiplications and counits of the comonoids.

Again, it follows from centrality that a comonoid map preserves multiple application of comultiplication and counits. Given a premonoidal category $\mathcal{K}$, comonoids and comonoid maps in $\mathcal{K}$ form a category $\mathbf{Comon}(\mathcal{K})$ with composition given by that of $\mathcal{K}$. Moreover, any strict premonoidal functor sends a comonoid to a comonoid, so any strict premonoidal functor $H : \mathcal{K} \longrightarrow \mathcal{L}$ lifts to a functor $\mathbf{Comon}(H) : \mathbf{Comon}(\mathcal{K}) \longrightarrow (\mathcal{L})$.

Trivially, any comonoid $\mathbf{C}$ in a premonoidal category $\mathcal{K}$ yields a comonad on $\mathcal{K}$ given by $- \otimes \mathbf{C}$, and any comonoid map $f : \mathbf{C} \longrightarrow \mathbf{D}$ yields a map of comonads from $- \otimes \mathbf{C}$ to $- \otimes \mathbf{D}$, and hence a functor from $\mathbf{Kleisli}(- \otimes \mathbf{D})$, the Kleisli category of the comonad $- \otimes \mathbf{D}$, to $\mathbf{Kleisli}(- \otimes \mathbf{C})$, that is the identity on objects. So we have a functor from $\mathbf{Comon}(\mathcal{C})^{\mathrm{op}}$ to $\mathcal{C}at$, which we denote by $\mathsf{s}(\mathcal{K})$. See [PR97] for this construction and another application of it.

Now, given a category $\mathcal{C}$ with finite products, every object A of $\mathcal{C}$ has a unique comonoid structure, given by the diagonal and the unique map to the terminal object. So $\mathbf{Comon}(\mathcal{C})$ is isomorphic to $\mathcal{C}$.

Thus, given a category $\mathcal{C}$ with finite products, a premonoidal category $\mathcal{K}$, and a strict premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$, we have a functor $\kappa(J) : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ given by $\mathsf{s}(\mathcal{K})$ composed with the functor induced by J from $\mathcal{C} \cong \mathbf{Comon}(\mathcal{C})$ to $\mathbf{Comon}(\mathcal{K})$.

8.3 κ-categories

In this section, we introduce κ-categories, and show that the construction at the end of Section 8.2 yields an equivalence between premonoidal categories with added structure as we shall make precise, and κ-categories.

Hasegawa has decomposed the λ-calculus into two calculi, the κ-calculus, and the ζ-calculus [Has95]. This analysis arose from study of Hagino's categorical programming language. The idea of the κ-calculus, also known as the contextual calculus, is that it has product types on which its abstraction and reduction are constructed, and it can be regarded as a reformulation of the first-order fragment of simply-typed λ-calculus, but does not require the exponent types. We do not explicitly present the κ-calculus here. However, we do describe the notion of κ-category, which is a categorical analogue of the definition of κ-calculus. Further, we compare the notion of κ-category with that of symmetric premonoidal category with a extra structure. That relationship is one of the main theorems of the chapter, which we later extend to relate our two main models of continuations.

Given a small category $\mathcal{C}$, a functor from $\mathcal{C}^{\mathrm{op}}$ to Cat is called an *indexed category*, a natural transformation between two indexed categories is called an *indexed functor*. The notion of *indexed natural transformation* is definable too, and this gives us a evident notion of adjunction between indexed categories. In concrete terms, it amounts to an $\mathbf{Ob}\mathcal{C}$-indexed family of adjunctions, such that the units and counits are preserved by reindexing along each $f : A \longrightarrow B$. And given an indexed category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$, we denote by $H^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ the indexed functor for which $H^{\mathrm{op}}_A = (H_A)^{\mathrm{op}}$ with H^{op}_f defined by H_f.

We will need the definitions of H^{op} and adjunctions between indexed categories in later sections to extend the notion of a functor being self-adjoint on the left, as in the semantics for continuations with premonoidal structure used to model environments in Chapter4 to that of an *indexed* functor being self-adjoint on the left as in the semantics for continutations using κ-categories to model environments in Section 8.4. But now for our definition of κ-category.

8.3.1 Definition A *κ-category* consists of a small category $\mathcal{C}$ with finite products, together with an indexed category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ such that

- for each object A of $\mathcal{C}$, $\mathbf{Ob}H_A = \mathbf{Ob}\mathcal{C}$, and for each arrow $f : A \longrightarrow B$ in $\mathcal{C}$, the functor $H_f : H_B \longrightarrow H_A$ is the identity on objects
- for each projection $\pi : B \times A \longrightarrow B$ in $\mathcal{C}$, the functor H_π has a left adjoint L_B given on objects by $- \times A$
- *(the Beck-Chevalley condition)* for every arrow $f : B \longrightarrow B'$ in $\mathcal{C}$, the natural transformation from $L_B \circ H_{f \times id_A}$ to $H_f \circ L_{B'}$ induced by the adjointness is an isomorphism.

$$\begin{tikzcd}
H_{B' \times A} \arrow[r, "L_{B'}"] \arrow[d, "H_{f \times \mathrm{id}_A}"'] \arrow[dr, phantom, "\Longrightarrow"] & H_{B'} \arrow[d, "H_f"] \\
H_{B \times A} \arrow[r, "L_B"'] & H_B
\end{tikzcd}$$

We shall denote the isomorphism associated with the adjunctions given in the definition by

$$\kappa : H_{B\times A}(C, C') \cong H_B(C \times A, C').$$

A κ-category allows us to model the environments in the presence of continuations or other computational effects. Of course, modelling computational effects involves more structure than that of a κ-category: for continuations, it requires the assignment to each type τ of a type $\neg\tau$ that awaits an input of type τ. We shall study such structure in Section 8.4, where we shall define an indexed $\neg$-category. But here, we restrict out attention to modelling environments, and we shall pursue our leading example, that of continuations, later.

8.3.2 Proposition Given a κ-category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$, there is an indexed functor $\mathsf{inc} : \mathsf{s}(\mathcal{C}) \longrightarrow H$ as follows: for each A in $\mathcal{C}$, we have a functor from $\mathsf{s}(\mathcal{C}_A)$ to H_A. On objects, it is the identity. To define inc_1 on arrows, given $f : A \longrightarrow B$ in $\mathcal{C}$, consider the arrow $\iota_B : 1 \longrightarrow B$ in H_B corresponding under the adjunction to id_B in H_1. Applying H_f to it gives a map $H_f(\iota_B) : 1 \longrightarrow B$ in H_A, or equivalently, under the adjunction, a map from A to B in H_1. Define $\mathsf{inc}_1(f)$ to be that map.

$$\begin{tikzcd}
\mathbf{1} & A \arrow[r, "f"] & B & \mathbf{1} \\
A \arrow[d, "\mathsf{inc}_1(f)"'] & \mathbf{1} \arrow[d, "H_f(\iota_B)"'] & \mathbf{1} \arrow[d, "\iota_B"] & B \arrow[d, "\mathrm{id}_B"] \\
B & B & B \arrow[l, mapsto, "H_f"'] & B
\end{tikzcd}$$

This plus naturality determines the rest of the structure.

PROOF It is immediate that inc_1 preserves identities, and one can prove that it preserves composition: this follows by proving that for any map $f : A \longrightarrow B$ in $\mathcal{C}$ and any map $g : 1 \longrightarrow C$ in H_B, the map $H_f(g)$ corresponds to the composite in H_1 of $\mathsf{inc}_1(f)$ with the adjoint correspondent to g. Moreover, this yields a functor inc_A for every A, with naturality as required. $\Box$

Using proposition 8.3.2, we can exhibit the relationship between symmetric premonoidal categories with specified extra structure and κ-categories. This forms the basis for the first main result of the chapter, Prop 8.3.4. First, for the construction of a κ-category from a symmetric premonoidal category, we have

8.3.3 Proposition Given a small category $\mathcal{C}$ with finite products, a small symmetric premonoidal category $\mathcal{K}$ and an identity on objects strict symmetric premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$, the functor $\kappa(J) : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ is a κ-category.

PROOF It follows immediately from the construction of $\kappa(J)$ in Section 8.2 that for each object A of $\mathcal{C}$, we have $\mathbf{Ob}\kappa(J)_A = \mathbf{Ob}\mathcal{C}$, and that for each arrow $f : A \longrightarrow B$ in $\mathcal{C}$, the functor $\kappa(J)_f$ is the identity on objects. Moreover, the existence of the adjoints to each $\kappa(J)_\pi$ follows directly from the construction and the fact that $\mathcal{C}$ is symmetric. The Beck-Chevalley condition also follows directly from the construction. $\square$

Now, for the converse, giving our first main result of the chapter.

8.3.4 Proposition Let $\mathcal{C}$ be a small category with finite products. Given a κ-category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$, there are a symmetric premonoidal category $\mathcal{K}$ and an identity on objects strict symmetric premonoidal functor $J : \mathcal{C} \longrightarrow \mathcal{K}$, unique up to isomorphism, for which H is isomorphic to $\kappa(J)$.

Proof Define $\mathcal{K}$ to be H_1. For each object A of $\mathcal{K}$, equally A an object of $\mathcal{C}$ since $\mathbf{Ob}H_1 = \mathbf{Ob}\mathcal{C}$, define $- \otimes A : \mathcal{K} \longrightarrow \mathcal{K}$ by the composite $L \circ H_!$ where $! : A \longrightarrow 1$ is the unique map in $\mathcal{C}$ from A to 1. Note that ! is of the form π, so the left adjoint exists. Moreover, for each map $g : C \longrightarrow C'$ in $\mathcal{K}$, we have $g \otimes A : C \times A \longrightarrow C' \times A$. The rest of the data and axioms to make $\mathcal{K}$ a symmetric premonoidal category arise by routine calculation, using the symmetric monoidal structure of $\mathcal{C}$ determined by its finite product structure, and by use of the Beck-Chevalley condition.

Define $J : \mathcal{C} \longrightarrow \mathcal{K}$ by inc_1 as in proposition 8.3.2. It follows from the Beck-Chevalley condition that for a map $f : A \longrightarrow B$ in $\mathcal{C}$, and for a map $g : C \longrightarrow D$ in H_B, we have that $H_f(g)$ is given by the composite of $J(id_C \times f)$ with the adjoint correspondent of g. The Beck-Chevalley condition further implies that $(\mathsf{inc}_1 -) \otimes A$ agrees with $\mathsf{inc}_1(- \times A)$. It follows from functoriality of the H_f's that every map in $\mathcal{C}$ is sent into the centre of $\mathcal{K}$. Functoriality plus the Beck-Chevalley condition similarly imply that all the structural maps are preserved. So J is an identity on objects strict symmetric premonoidal functor.

It follows directly from our construction of J that $\kappa(J)$ is isomorphic to H. Moreover, $J : \mathcal{C} \longrightarrow \mathcal{K}$ is fully determined by H since $\mathcal{C}$ is fixed, $\mathcal{K}$ must be H_1 up to isomorphism, with premonoidal structure as given, and J must agree on maps with the construction as we have given it. Hence, J is unique up to isomorphism. $\square$

8.4 Continuation semantics in indexed $\neg$-categories

In this section, we use the definition of κ-category as a basis, together with self-adjointness, for defining the notion of an indexed $\neg$-category. We then use that latter definition to give our third continuations semantics. In the final section, we shall prove that it is essentially equivalent to the second, i.e., that given by $\otimes\neg$-categories.

8.4.1 Definition An *indexed* $\neg$*-category* consists of a κ-category $H : \mathcal{C}^{\mathrm{op}} \longrightarrow \mathcal{C}at$ together with an indexed functor $\neg : H^{\mathrm{op}} \longrightarrow \mathsf{s}(\mathcal{C})$ such that $\mathsf{inc} \circ \neg$ is self-adjoint on the left, together with a coretract thunk of $\mathsf{force}_{\mathbf{1}} \circ \mathsf{inc}$, where force is the unit of the self-adjunction, such that

- force is dinatural in A with respect to all maps in $H_{\mathbf{1}}$ and
- letting $(\underline{\underline{\mathsf{force}}}_A)_B$ be the correspondent under the adjunction to $L_A H_\pi((\mathsf{force}_{\mathbf{1}})_B)$, we have

$$\begin{array}{rcl}
\neg\mathsf{force}_{\mathbf{1}} & = & \mathsf{thunk}_\neg \\
\mathsf{thunk}; \neg\neg\underline{\underline{\mathsf{force}}} & = & \underline{\underline{\mathsf{force}}}; \mathsf{thunk} \\
\mathsf{thunk}_{A\times C} & = & A \times \mathsf{thunk}_C; A \times \neg\underline{\underline{\mathsf{force}}}; \underline{\underline{\mathsf{force}}} \\
\neg\kappa^{-1}(\mathsf{id}_{C\times A}) & = & \neg_C(LH_!(\mathsf{force}_{\mathbf{1}})); \mathsf{force}_C
\end{array}$$

The left adjoint to reindexing along projections gives rise to a comonad on each fibre, which we will write as $(\_) \otimes A$. Furthermore , using inc, we have a diagonal map $\delta_A : A \longrightarrow A \otimes A$ in each fibre.

The thinking behind the definition is as follows. The category $\mathcal{C}$ with its finite product structure allows us to model an environment as the product of the types it contains. In the indexed category, program phrases defined in an environment will be modelled as elements in the fibre over the denotation of that environment.

$$[\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$$

$$\begin{tikzcd}
\mathbf{1} \arrow[d, "{[\![x_1:\sigma_1,\ldots,x_n:\sigma_n \vdash M:\tau]\!]}"] \\
{[\![\tau]\!]}
\end{tikzcd}$$

The isomorphism of adjunction κ is a first order binding construct that allows us to make the dependency of a program phrase on certain variables explicit. The negation functor is much as before, except that it now acts on those variables explicitly singled out by a previous κ.

$$\begin{tikzcd}
\Gamma \times C & \Gamma & \Gamma & \Gamma \\
A \arrow[d, "f"] \arrow[r, phantom, "f \overset{\kappa}{\longmapsto} \kappa f"] & A \times C \arrow[d, "\kappa f"'] & A \arrow[d, "f"] \arrow[r, phantom, "f \overset{\neg}{\longmapsto} \neg f"] & \neg B \\
B & B & B & \neg A \arrow[u, "\neg f"']
\end{tikzcd}$$

The motivation for the axioms is as for $\otimes\neg$-categories, except that here, we can avoid one of the axioms as it follows from the indexing of $\neg$. However, we need our last axiom here in order to make the indexing of $\neg$ coherent: intuitively, it means that negating the retrieving of a value of type C from the environment to cons a value of type C to a value of type A gives us an operation of partially satisfying demand for a value of type C while leaving the demand for a value of type A untouched.

This formalism, unlike that for a $\otimes\neg$-category, separates the data and the control mechanisms. The indexed functor $\neg$ is in some sense oblivious to the indexed structure with which first order data manipulation is described. We do not want control to interfere with any data with which it is not concerned. So the ability to model continuations with indexed categories as we do here is a clear indication that we have separated the two. In the final section, we show that this modelling is essentially equivalent to that using premonoidal categories and self-adjointness. We take this as evidence that modelling continuations by self-adjointness is a robust notion in the sense that it is not overly sensitive to the way we model environments, as we could model them in two different ways, in each case fitting the self-adjointness into this framework.

To model λ+`callcc`, types are interpreted as objects in $\mathcal{C}$. Environments are interpreted using the product in $\mathcal{C}$.

$$\begin{array}{rcl}
[\![\neg\tau]\!] & \stackrel{\text{def}}{=} & \neg[\![\tau]\!] \\
[\![\sigma \to \tau]\!] & \stackrel{\text{def}}{=} & \neg([\![\sigma]\!] \otimes \neg[\![\tau]\!]) \\
[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] & \stackrel{\text{def}}{=} & [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]
\end{array}$$

A judgement $\Gamma \vdash M : \tau$ denotes an element $[\![\Gamma \vdash M : \tau]\!] : \mathbf{1} \longrightarrow [\![\tau]\!]$ in the fibre over $[\![\Gamma]\!]$.

$$\begin{array}{rcl}
[\![x_1 : \tau_1, \ldots x_n : \tau_n \vdash x_j : \tau_j]\!] & \stackrel{\text{def}}{=} & H_{\pi_j}\kappa^{-1}(\mathsf{id}_{[\![\tau_j]\!]}) \\
[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] & \stackrel{\text{def}}{=} & \mathsf{thunk}; \neg(\kappa\neg[\![\Gamma, x : \sigma \vdash M : \tau]\!]) \\
[\![\Gamma \vdash \mathtt{throw}\ M\ N : \sigma]\!] & \stackrel{\text{def}}{=} & [\![\Gamma \vdash M : \neg\tau]\!]; \neg(\kappa(H_{\pi_1}[\![\Gamma \vdash N : \tau]\!])); \mathsf{force} \\
[\![\Gamma \vdash \mathtt{callcc}\ M : \tau]\!] & \stackrel{\text{def}}{=} & [\![\Gamma \vdash M : \neg\tau \to \tau]\!]; \neg\delta; \mathsf{force} \\
[\![\Gamma \vdash MN : \tau]\!] & \stackrel{\text{def}}{=} & [\![\Gamma \vdash M : \sigma \to \tau]\!]; \neg([\![\Gamma \vdash N : \sigma]\!] \otimes \neg[\![\tau]\!]); \mathsf{force}
\end{array}$$

Again, the semantics as such is not the topic of the present chapter. We only give some hint at how it is intended to work.

We write a morphism from X to Y in the fibre over C as

$$X \xrightarrow[C]{} Y$$

The most interesting clause is the one for λ-abstraction in that abstracting over a variable implies moving from one fibre to another.

In a more traditional (call-by-name) setting, λ would be interpreted by means of an adjoint to reindexing. Here, it is more elaborate, as it is decomposed into the first-order abstraction given by the structure on the fibration on the one hand and the fibrewise "negation" given by the continuation functor on the other.

A judgement $\Gamma, x : \sigma \vdash M : \tau$ denotes a morphism

$$\mathbf{1} \xrightarrow[\llbracket\Gamma\rrbracket\times\llbracket\sigma\rrbracket]{} \llbracket\tau\rrbracket$$

Negating this given an arrow

$$\neg\llbracket\tau\rrbracket \xrightarrow[\llbracket\Gamma\rrbracket\times\llbracket\sigma\rrbracket]{} \neg\mathbf{1}$$

which, by virtue of κ, amounts to

$$\llbracket\sigma\rrbracket \times \neg\llbracket\tau\rrbracket \xrightarrow[\llbracket\Gamma\rrbracket]{} \neg\mathbf{1}$$

Negating this yields a morphism

$$\neg\neg\mathbf{1} \xrightarrow[\llbracket\Gamma\rrbracket]{} \neg(\llbracket\sigma\rrbracket \times \neg\llbracket\tau\rrbracket)$$

All that remains to be done in order to get the meaning of $\lambda x.M$ is to precompose with $\mathsf{thunk} : \mathbf{1} \longrightarrow \neg\neg\mathbf{1}$, taking care of the double negation:

$$\mathbf{1} \xrightarrow[\llbracket\Gamma\rrbracket]{} \neg\neg\mathbf{1} \xrightarrow[\llbracket\Gamma\rrbracket]{} \neg(\llbracket\sigma\rrbracket \times \neg\llbracket\tau\rrbracket)$$

8.5 Relating $\otimes\neg$-categories and indexed $\neg$-categories

In this final section of the chapter, we build upon the equivalence between κ-categories and symmetric premonoidal categories with the extra structure specified in Proposition 8.3.4 to relate $\otimes\neg$-categories and indexed $\neg$-categories. They are almost but not quite equivalent. The only difference lies implicit in Proposition 8.3.4: for our definition of $\otimes\neg$-category, we assert that the centre of our category has finite products, whereas Proposition 8.3.4 merely asserts that we have a category with finite products mapping, as the identity on objects, into the centre of our category. We regard this as a minor difference, as the latter merely extends the former mildly without changing any other structure.

Let $\delta_A \stackrel{\text{def}}{=} \langle \mathsf{id}_A, \mathsf{id}_A \rangle : A \longrightarrow A \otimes A$

Let $\mathcal{K}$ be a $\otimes\neg$-category. Let $*_A$ be the Kleisli composition

$$f *_A g \stackrel{\text{def}}{=} \langle \pi_1, \mathsf{id} \rangle ; A \otimes f \otimes g$$

Define $\neg_A : \mathbf{Kleisli}_{\mathcal{K}}(A \otimes (\_))^{\mathrm{op}} \longrightarrow \mathbf{Kleisli}_{\mathsf{Z}(\mathcal{K})}(A \times (\_))$ by $\neg_A B = B$ on objects and by

$$\neg_A f \stackrel{\mathrm{def}}{=} A \otimes \neg f; \underline{\underline{\mathsf{apply}}}_A$$

on morphisms. This is well-defined: $A \otimes \neg f$ is central, because $\neg f$ is, and

$$\underline{\underline{\mathsf{apply}}} = \underline{\underline{\mathsf{apply}}}; \mathsf{thunk}; \neg\mathsf{thunk} = \mathsf{thunk}; \neg\neg\underline{\underline{\mathsf{apply}}}; \neg\mathsf{thunk}$$

is also central.

Define $\mathsf{force}_A : \neg_A\neg_A B \longrightarrow B$ in $\mathbf{Kleisli}_{\mathcal{K}}(A \otimes (\_))$ by $\mathsf{force}_A \stackrel{\mathrm{def}}{=} \pi_2; \mathsf{apply}_{\mathbf{1}}$.

$\neg_A$ preserves identities $\pi_2 : A \otimes B \longrightarrow B$ because

$$\begin{aligned}
& \neg_A(\pi_2) \\
= \;& \neg_A(! \otimes B) \\
= \;& A \otimes \neg(! \otimes B); A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A \\
= \;& A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; ! \otimes B); \mathsf{apply}_A \\
= \;& A \otimes \neg(! \otimes \neg\neg B; \mathbf{1} \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A \\
= \;& A \otimes \neg(\mathbf{1} \otimes \mathsf{apply}_{\mathbf{1}}); A \otimes \neg(! \otimes \neg\neg B); \mathsf{apply}_A \\
= \;& A \otimes \neg(\mathbf{1} \otimes \mathsf{apply}_{\mathbf{1}}); ! \otimes \neg(\mathbf{1} \otimes \neg\neg B); \mathsf{apply}_{\mathbf{1}} \\
= \;& A \otimes \neg(\mathsf{apply}_{\mathbf{1}}); ! \otimes \neg\neg\neg B; \mathsf{apply}_{\mathbf{1}} \\
= \;& ! \otimes \neg B; \mathbf{1} \otimes \neg(\mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_{\mathbf{1}} \\
= \;& ! \otimes \neg B \\
= \;& \pi_2 : A \otimes \neg B \longrightarrow \neg B
\end{aligned}$$

$\neg_A$ preserves composition: let $f : A \otimes B \longrightarrow C$ and $g : A \otimes C \longrightarrow D$. Then

$$\begin{aligned}
& \neg_A(f *_A g) \\
= \;& \neg_A(\delta_A \otimes B; A \otimes f; g) \\
= \;& A \otimes \neg(\delta_A \otimes B; A \otimes f; g); A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}); \mathsf{apply}_A \\
= \;& A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; \delta_A \otimes B; A \otimes f; g); \mathsf{apply}_A \\
= \;& \ldots \\
= \;& \delta_A \otimes A \otimes A \otimes \neg C; A \otimes A \otimes \neg g; A \otimes A \otimes \neg(A \otimes f); \underline{\underline{\mathsf{apply}}}_{A \otimes A}
\end{aligned}$$

$$\begin{aligned}
& \neg_A(f) *_A \neg_A(g) \\
= \;& \delta_A \otimes \neg B; A \otimes A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; f); A \otimes \mathsf{apply}_A; A \otimes \neg(A \otimes \mathsf{apply}_{\mathbf{1}}; g); \mathsf{apply}_A \\
= \;& \ldots \\
= \;& \delta_A \otimes A \otimes A \otimes \neg C; A \otimes A \otimes \neg g; A \otimes A \otimes \neg(A \otimes f); A \otimes \underline{\underline{\mathsf{apply}}}_A; \underline{\underline{\mathsf{apply}}}_A
\end{aligned}$$

So the required identity follows from the axiom

$$\mathsf{apply}_{A\otimes A'} = \langle \pi_2, \pi_1 \rangle \otimes \neg(A \otimes A' \otimes \neg B); A' \otimes \underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_{A'}$$

and the facts that $\langle \pi_2, \pi_1 \rangle$ is central and $\delta; \langle \pi_2, \pi_1 \rangle = \delta$.

The triangular identity $\neg_A \mathsf{force}_A *_A \mathsf{force}_A = \mathsf{id}$ holds:

$$\begin{array}{rl} & \neg_A \mathsf{force}_A *_A \mathsf{force}_A \\ = & \neg_A(\pi_2; \mathsf{apply}_{\mathbf{1}}) *_A \mathsf{force}_A \\ = & (A \otimes \neg\mathsf{apply}_{\mathbf{1}}; \neg_A(\pi_2)) *_A \mathsf{force}_A \\ = & (A \otimes \neg\mathsf{apply}_{\mathbf{1}}; \pi_2) *_A \mathsf{force}_A \\ = & \delta_A \otimes \neg B; A \otimes A \otimes \neg\mathsf{apply}_{\mathbf{1}}; A \otimes \pi_2; \pi_2; \mathsf{apply}_{\mathbf{1}} \\ = & A \otimes \neg\mathsf{apply}_{\mathbf{1}}; \delta_A \otimes \neg\neg\neg B; A \otimes \pi_2; \pi_2; \mathsf{apply}_{\mathbf{1}} \\ = & A \otimes \neg\mathsf{apply}_{\mathbf{1}}; \pi_2; \mathsf{apply}_{\mathbf{1}} \\ = & A \otimes \neg\mathsf{apply}_{\mathbf{1}}; ! \otimes \neg\neg\neg B; \mathsf{apply}_{\mathbf{1}} \\ = & ! \otimes \neg B; \mathbf{1} \otimes \neg\mathsf{apply}_{\mathbf{1}}; \mathsf{apply}_{\mathbf{1}} \\ = & \pi_2; \neg\mathsf{apply}_{\mathbf{1}}; \mathsf{apply}_{\mathbf{1}} \\ = & \pi_2 : A \otimes \neg B \longrightarrow \neg B \end{array}$$

force is natural:

$$\begin{array}{rl} & \neg_A \neg_A f *_A \mathsf{force}_A \\ = & A \otimes \neg(A \otimes \neg f; \underline{\underline{\mathsf{apply}}}_A); \underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_{\mathbf{1}} \\ = & A \otimes \neg\underline{\underline{\mathsf{apply}}}_A; A \otimes \neg(A \otimes \neg f); \mathsf{apply}_A \\ = & A \otimes \neg\underline{\underline{\mathsf{apply}}}_A; \mathsf{apply}_A; f \\ = & A \otimes \mathsf{apply}_{\mathbf{1}}; f \\ = & \mathsf{force}_A *_A f \end{array}$$

Putting this all together, it follows that we have

8.5.1 Proposition Given a $\otimes\neg$-category, $(\mathcal{K}, \neg, \mathsf{apply}, \mathsf{thunk})$, the construction $(\kappa(J), \neg_A, \mathsf{force}_A)$ as above, together with the given thunk, give an indexed $\neg$-category.

PROOF Most of the proof is given above. For the rest, the axioms hold simply because the category $H_{\mathbf{1}}$ is given by $\mathcal{K}$. □

8.5.2 Proposition Given a symmetric premonoidal category $\mathcal{K}$ for which the premonoidal structure restricts to finite product structure on the centre, to extend this to the structure of a $\otimes\neg$-category is equivalent to extending the structure of the κ-category $\kappa(J)$ to that of an indexed $\neg$-category.

PROOF We need to prove that the construction of the proposition is a bijection up to isomorphism. Given an indexed $\neg$-category, one can obtain a $\otimes\neg$-category by considering $H_{\mathbf{1}}$. In order to show that the construction applied to that $\otimes\neg$-category yields the original indexed $\neg$-category, everything is routine provided one can show that for any indexed $\neg$-category, the behaviour of $\neg$ on $H_{\mathbf{1}}$ determines its behaviour on H_A for all A. But this follows from the fact that $\neg$ is indexed and from the final axiom. $\square$

There is little difference between the notions of indexed $\neg$-category and $\otimes\neg$-category. The only difference between them lies in the choice of an explicitly given category with finite products and an identity on objects strict monoidal functor into a symmetric premonoidal category rather than consideration of a property of the centre. The former is the structure given naturally by an indexed $\neg$-category. Computationally, it is natural to assume that in the presence of first-class continuations the whole of the centre admits finite products. This is because the self-adjoint structure allows every central morphism to be reified, as explained in section 6.3.

Chapter 9

Towards a graphical representation of CPS

In this chapter, we present a graphical notation that may be seen as an extreme distillation of CPS (a negation-only fragment). This graphical representation relates to CPS roughly as Milner's graphical action structure PIC [Mil93] for the π-calculus relates to the full π-calculus. We may regard it as giving some insight, though the match with CPS is not a perfect one. (See also [Mil94] and [Par95].)

Given the composition and identity definable in the calculus, we naturally arrive at a ("CPS") monoid. Despite its simplicity it has some of the deeper structure characteristic of CPS: considered as a one-object category, it comes equipped with a contravariant functor self-adjoint on the left and on the right.

Among the aspects of CPS that can be illustrated by the graphical presentation we would like to point out the following:

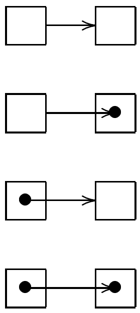
- The self adjointness, and in particular
- a view of the isomorphism of adjunction as turning a program upside down
- A view of variables as nodes in a graph or pointers

While our graphical formalism allows to visualise the above, it fails to address other aspects of CPS. The self-adjointness "degenerates" (in the sense of a line degenerating to a point, say) to a duality. However, the degeneracy is not a collapse (in the sense that all morphisms are equal).

9.1 A graphical calculus

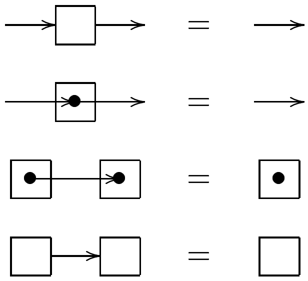
The $\boxed{\bullet}\!\!\longrightarrow$ -calculus consists of boxes $\boxed{}$, possibly containing "bullets" $\bullet$, linked by directed edges $\longrightarrow$.

An arrow can be linked (on either end) to the box or the bullet it contains; hence there are four possible ways (apart from the direction of the arrow) in which two boxes can be linked.



If nothing connects to the bullet, we omit it.

The calculus has the following four rules:



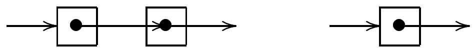
(α)

(β)

(η)

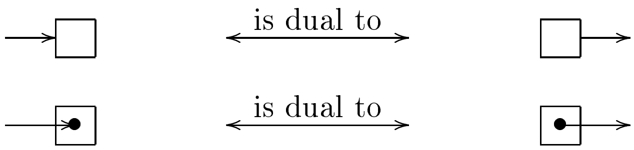
(σ)

Strictly speaking, these rewrite rules are only a shorthand for a more complicated pattern matching. In the last two laws, the two boxes are to be fused into one; this (and if there is one, the • inside) inherits all arrows connected to either of the fused boxes. For example, applying η may look like this:

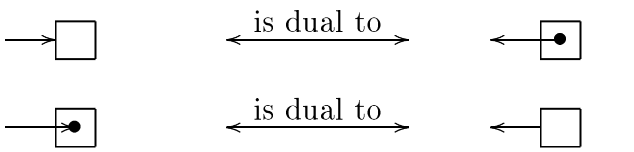


9.2 Duality, or inside out

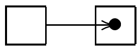
We point out two dualities. Poincaré duality: the rôles of boxes and arrows are interchanged, while the bullet/box distinction remains:



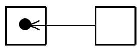
While the Poincaré duality appears to reflect a certain symmetry of the formal set-up, the second duality, that between boxes and bullets may be more relevant as an operation.



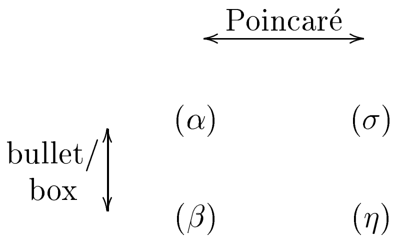
This latter duality can be regarded as “turning inside out” inasmuch as the dual of putting the left box into the one on the right



is given by putting the right box into the left one



The rules are connected by the dualities like this:

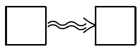


So up to the two dualities, the four laws are only a single law, stating that connected things at the same level of box nesting can be merged.

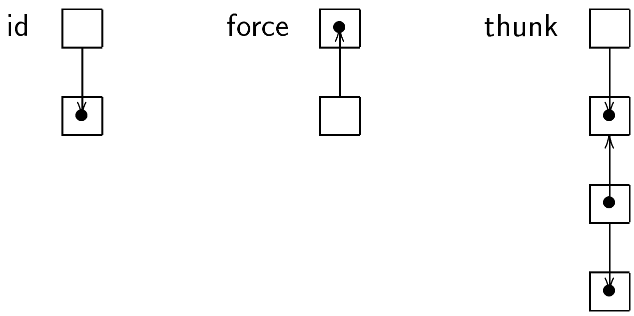
9.3 The CPS monoid

The elements of the CPS monoids are finite nonempty sequences of boxes, where any two adjacent ones are connected in one of the eight possible ways, i.e. any box/bullet combination and any direction of the connecting arrow. We usually draw these vertically or, to save space, from left to right. Elements which can be proved equal using the laws are identified (so the elements of the monoid are actually equivalence classes, but this will be glossed over by representative-wise definitions etc.).

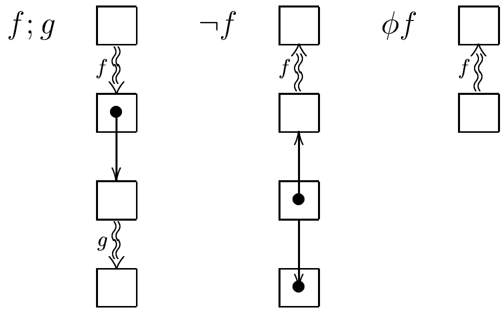
We use



as a meta-notation ranging over morphisms. Here are some examples of morphisms that will be used later.

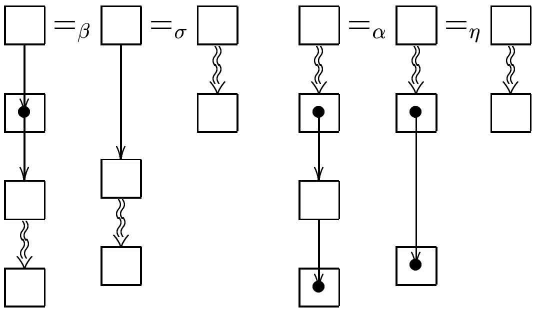


In addition to these constants, we also have operations on morphisms. For morphisms f and g, the composition $f; g$, the negation $\neg f$ and the transpose ϕf are defined as follows:



The associativity of composition is trivial.

We write proofs of equations about the CPS monoid as sequences of graphs side by side, with adjacent graphs transformed into each other by one rewrite step. id is the identity:

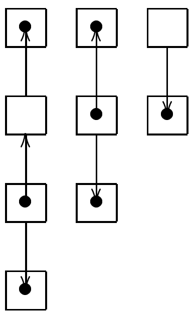


9.4 Self-adjointness, or upside down

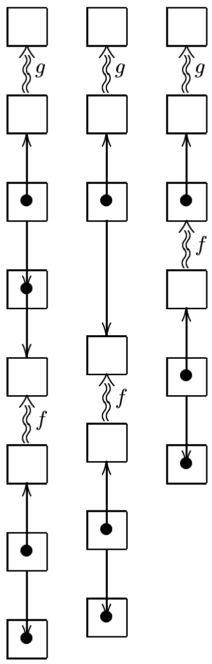
9.4.1 Proposition $\neg$ is a contravariant functor right adjoint to its dual. force is both the unit and counit of this adjunction.

PROOF

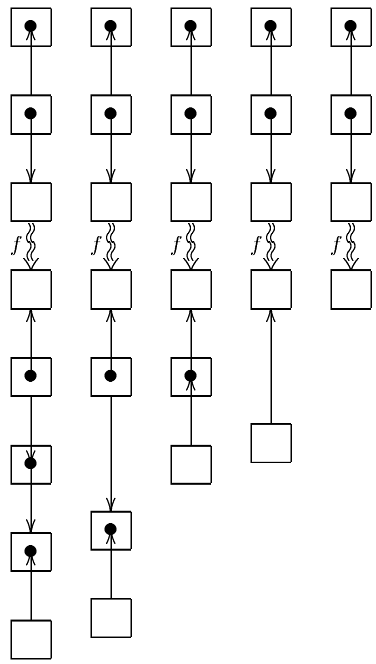
$\neg$ preserves identities $\neg\mathsf{id} = \mathsf{id}$:



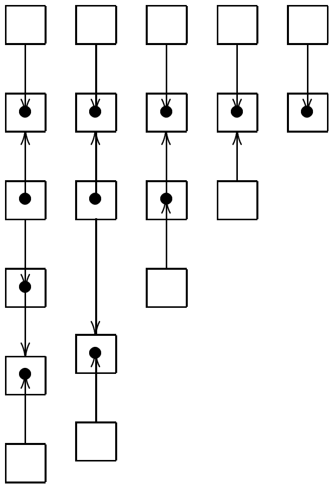
¬ preserves composition $\neg g; \neg f = \neg(f; g)$:



¬ is natural: $\neg\neg f; \mathsf{force} = \mathsf{force}; f$



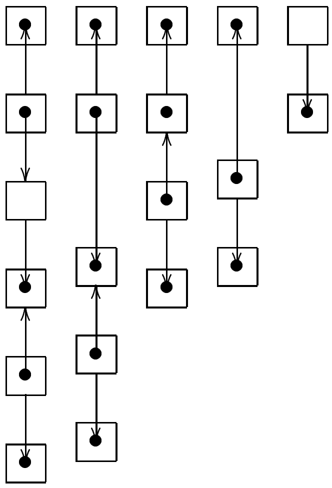
triangular identity for $\neg$: $\neg\mathsf{force};\mathsf{force} = \mathsf{id}$



□

This completes the proof of the adjointness. Notice that the adjoint correspondent of a morphism f is just its transpose ϕf, that is, f upside down.

The self-adjointness on the right follows from the fact that force and thunk are actually *inverses* in this model, i.e. $\mathsf{force};\mathsf{thunk} = \mathsf{id}$. Hence force is the unit of a duality; but then so is its inverse thunk.

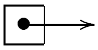


9.5 A semantics for linear unary CPS calculus

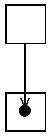
Recall the typing of linear unary CPS calculus.

$$\frac{}{xk \vdash k\langle x\rangle} \qquad \frac{}{xk \vdash x\langle k\rangle}$$

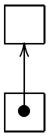
$$\frac{xn \vdash M \quad yk \vdash N}{xk \vdash M\{n\langle y\rangle{=}N\}} \qquad \frac{nk \vdash M \quad yx \vdash N}{xk \vdash M\{n\langle y\rangle{=}N\}}$$

We can give a semantics for the linear unary CPS-calculus in the  -monoid

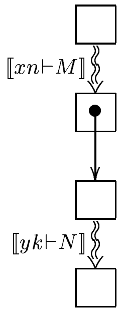
$[\![xk \vdash k\langle x\rangle]\!]$



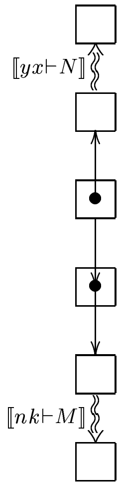
$[\![xk \vdash x\langle k\rangle]\!]$



$[\![xk \vdash M\{n\langle y\rangle{=}N\}]\!]$



$[\![xk \vdash M\{n\langle y\rangle{=}N\}]\!]$



This semantics can be summed up in the following recipe for drawing the graphical representation of a term. To translate a linear unary CPS term, do the following:—

- α-convert if necessary, making all variables pairwise distinct;
- draw a box for each variable;
- for every subterm of the form $k\langle x\rangle$, draw an arrow from the box for x to a bullet in the box for k;
- for every subterm of the form $M\{n\langle x\rangle{=}N\}$, draw an arrow from a bullet in the box for N to the box for x.

9.6 Duality and degeneracy

Put crudely, the graphical representation can account for $\ldots\langle\ldots\rangle$, but not $\ldots\{\ldots = \ldots\}$. This is analogous to the way PIC does not account for guarding [Mil94].

The self-adjointness and the duality appear closely connected.

Somewhat more ominously, we have

$$\mathsf{force};\mathsf{thunk} = \mathsf{id}$$

This means that force and thunk are both isomorphisms; hence both self-adjointnesses collapse to a duality.

Thus this calculus is a degenerate model even of the tiny fragment of CPS that it can describe, as it does not adequately account for reification. It is reification (wrapping things into closures) that makes a proper CPS negation non-involutive. On the other hand, the calculus has a great deal of the intuitive flavour of CPS; and if the missing ingredient is indeed only reification, we could hope for the following connection between CPS negation (self-adjointness, unit not iso) and classical negation (duality):

$$\text{CPS} - \text{reification} = \text{duality} = \text{classical negation}$$

If this is so, one should get a classical negation from CPS by adding a law for dissolving closures:

$$\text{classical negation} = \text{CPS} + (-\text{reification})$$

So the degeneracy of this model may actually be useful for exploring those connections in a simplified setting. Note that attempting to add extra axioms to the CPS calculus in order to enforce "classicality" will quite easily lead to collapse, e.g. if one makes $\mathsf{force}; \mathsf{thunk} = \mathsf{id}$ by adding the following axiom

$$h\langle k\rangle\{k\langle a\rangle{=}n\langle f\rangle\{f\langle p\rangle{=}p\langle a\rangle\}\} = n\langle h\rangle \qquad (\textsc{inv})$$

This issue of the degeneracy of a self-adjointness (for instance to a duality), is also explored in Masahito Hasegawa's manuscript [Has97].

Preliminary though they are, the ideas in this chapter may lead to two complementary directions for further work. On the one hand, we could attempt to add extra structure on the graphs to represent reification. A conventional way to do this would be to add boxes to encapsulate certain subgraphs and preclude certain reductions, as in [Mil94] and [Par95].

On the other hand, the duality aspects of CPS may become clearer if addressed in a setting where there actually is a *duality functor*. Duality seems to have a powerful, though somewhat ambiguous, influence on intuitions about continuations. (For instance, the subtitle of Filinski's early categorical account of continuations is "an investigation of duality".)

A related point is the duality on terms from section 3.6, which can be visualized here as the box/bullet duality.

Chapter 10

Conclusions and directions for further research

10.1 Conclusions

We have tried to show that continuations are a universal raw material from which low-level as well as high-level programming language constructs can be crafted. The categorical properties of the operations, such as force basically being the unit of adjunction, appear to defy common prejudices against control manipulation as excessively low-level and unprincipled. Like functions, continuations scale up well, but unlike functions, they do not require "purity". That is to say, the adjunction is not destroyed by the addition of effects in the style done by call-by-value languages like ML or Scheme.

To some extent what we have attempted here is a bridge-building effort between a tradition of semantics, centred around a few institutions in the USA, relying much more on metacircular interpreters for Scheme than on, say, domains and the more mathematically inclined European (predominantly British) tradition. Whether or not the Schemers need anyone to tell them that there is an adjointness about remains to be seen, but we believe that for the European tradition of semantics, it is of particular importance to maintain the link between theoretical fields such as categorical semantics and programming languages.

10.2 Directions for further work

10.2.1 Language design

The self-adjointness seems characteristic of continuations that are first-class as well as statically bound. Neither ML-style exceptions nor a Lisp-style (dynamically bound) `catch` construct appear to give rise to this kind of structure. We conjecture, then, that self-adjointness is a semantic criterion that sets first-class continuations apart.

Perhaps the best use of category theory in the semantics of programming languages is to rationalise semantic definitions [Ten91]; we hope to add more objective reasons to the general impression that continuations are, for lack of a better word, "cool". (This may even lend some moral support to the cause of their inclusion in future ML-like languages, whether idealised or real.)

The treatment of continuations in the type system of Standard ML of New Jersey is already fairly conducive to our semantic views of them (not least because it shaped them in the first place). A suggestion that we could offer is to facilitate passing between functions and continuations, by making it easy to convert between, or even identifying arrow types $\sigma \to \tau$ with special continuations, i.e. $\neg(\sigma * \neg\tau)$.

In traditional imperative languages, both functions and continuations (in the form of jump labels, say) are very far from being first-class: both can be introduced only as compile-time literals.

But semantically (even more categorically) it is precisely the first-class version that is more natural. First-class functions (of different flavours) give rise to cartesian, or monoidal, or central, closure. First-class continuations give self-adjointness.

On the basis of that, we would argue that `callcc` is the natural choice of control operator. While advanced, it is not particularly incomprehensible. Attempts to "improve" on it may be counterproductive.

The drive for first-class notions without arbitrary restrictions (as well as the related principle of orthogonality of such notions), appears to be one of the major feedbacks from semantics into practice, in particular language design, and on a more everyday level, programming language teaching.

(Given that first-class control is more natural than plain `goto` in that it enjoys universal properties, one could be led to speculate whether there is not a notion of *first-class* state more natural than plain "`:=`".)

10.2.2 Applications to programming

While Scheme in particular is expression-oriented, the categorical combinators could form the basis for a more composition-oriented approach to continuations.

We conjecture that they, together with a recursive (reflexive?) continuation type, could be used as a set of primitives for upward (in the sense of [FWH92]) continuations and coroutines in particular. (See also sections 1.1.2 and 1.2) in the introduction.

A related point is that we seem to be almost forced to define a continuation transformer akin to the negation functor for reasoning about `callcc` in the setting of an ML style typing of continuations. See 7.1.1.

10.2.3 Relation to π- and related calculi

Continuation Passing Style transformations have been studied primarily as a translation between different versions of λ calculus. Such a translation between different λ-calculi is indifferent to the calling mechanism of the target calculus. What appears to have been studied much less (although it underlies CPS compiling) is that CPS is indifferent even as to whether the target language is a λ-calculus at all, in that function application is translated into a kind of message passing between caller and callee. In that sense, CPS transforms are closer in spirit to π-calculus and related formalisms than to λ-calculi.

Much of the expressiveness of the π-calculus appears to be due to the fact that its "first-class" names can be used to implement generalized first-class continuations. We propose to find criteria for when names are used in such a CPS discipline, in order to scale up some of the essentially simple structure of CPS from the sequential to a concurrent scenario. In particular, a categorical characterization of continuation types appears to be quite robust in the sequential case, so that a generalisation to channels used as continuations appears possible. This would allow the isolation of a class of well-behaved computations lying properly between the purely functional and the fully concurrent. The ability of the languages in this class to accommodate (generalized) continuations would provide a more modular account of their expressive power. This would automatically entail the existence of encodings (CPS transforms) of various evaluation strategies for the λ-calculus, with control operators for free.

A related use of CPS as a unifying principle would be to take the existence of a CPS transform between two languages as a comparison of their expressiveness. We conjecture that a CPS discipline can be found not only in λ-calculus encodings [Bou97], but also in Sangiorgi's translation of Higher Order π-calculus back into the π-calculus.

We hope to explore these connections and extend the analysis of CPS as a particularly structured form of name-passing. Dually, given the popularity of graphical representations in concurrency, aspects of CPS could perhaps be elucidated by building on graphical presentations of, say, the π-calculus. Other techniques from concurrency that could profitably be conferred on CPS include contextual notions of equivalence and bisimulation.

Among the uses of CPS in compiling, we may distinguish between optimisations and translations to more low-level constructs.

The former do not change the character of the language: they re-write λ-terms into other, more efficient, λ-terms. The latter eliminate the λ altogether in favour of jumps with arguments.

In [FSDF93], it is argued that CPS is superfluous for optimisation purposes, as one can optimise while staying in the source language by so-called A-reductions that allow the same optimisations to be made as after a CPS transform. That would imply that CPS transforms, if considered as transforms from the λ-calculus to itself, are of little use to compiling. That would shift the emphasis to the other aspect of CPS, which we have attempted to address here, namely breaking down the λ's, as it were.

10.2.4 The expressive power of `callcc`

Section 6.7 consitutes preliminary evidence to the discriminating power of tt callcc. It seems possible that the equivalence induced on λ-terms by the Plotkin CPS transform (M and N are equivalent iff $\underline{M} = \underline{N}$) is similarly fine-grained to that given by Milner's π-calculus encoding, which coincides with the Levy-Longo tree equality [San94]. Specifically, the presence of first-class continuations allows to distinguish values from general computations.

10.2.5 Internal languages

Both the concreteness and ease of use of the CPS calculus and the more abstract viewpoint based on the self-adjointness are useful, not least because they complement each other. Their relationship would be clarified if the CPS calculus were, in a precise sense, the internal language of a $\otimes\neg$-category.

Similarly, one could hope for a fuller development of the flattened calculi as the internal languages for premonoidal categories.

10.2.6 Robustness

We have argued informally that the structure that we propose for first-class continuations is not something specific to the small fragment of programming languages that we actually studied and that it would still be present in a more realistic language with state, I/O and other effects. A connection with state has been sketched in section 7.4.

10.2.7 Refinement of the standard model

As a first refinement of the standard model, we propose that some care should be taken in the choice of the result type R. Of course, literally any type in $\mathcal{C}$ could be taken as the result type, but it should be clear that the choice of the terminal object yields a trivial interpretation. Other choices, such as a NNO are more defensible and possibly sufficient for PCF-like languages [SF90].

As one of the fundamental properties of the CPS transform is its being polymorphic in the result type, we would argue that this should be reflected in the model.

Realizability models have enough structure to allow for such a polymorphic result type. In the terminology of [FT95], we could take the result type R to be the "generic predomain" $\begin{pmatrix} G \\ \downarrow \\ C_0 \end{pmatrix}$ in a slice of the category of assemblies. (While this is probably not the best account of a "generic" type is seems quite sufficient for our purposes here.) If we wanted to add recursion, we could simply take the lift of this object instead; that would just amount to its fibrewise lift.

It appears that the abort operator $\mathcal{A}$ could not be well typed in this setting. No value produced by a program phrase could be polymorphic enough, as it were, to inhabit the result type.

Another conjecture is that in this setting, the control-flow and the domain-theoretic concepts of lifting should coincide, as on grounds of parametricity one would expect, for any base type A, that

$$R_\perp^{R_\perp^A} \cong A_\perp$$

(The question arises: over how big a collection of types does R have to vary for this to hold?)

10.2.8 Relation to polymorphism and semantics in general

The difficulties faced when trying to accommodate continuations as morphisms with a domain but *without a codomain*, as it were, are in some ways similar to those when attempting to account for polymorphic functions as morphisms without *fixed* domain and codomain. In each case the mathematical framework seems ill-equipped to deal with them, being based on (monomorphic, non-escaping) functions.

There is also a more direct link to polymorphism in continuation semantics. If one does model continuations as functions, at least one should wish to do justice to the parametricity of the answer type and explore its ramifications. Examples of the latter are the existence or non-existence of the abort operator and control delimiters and the issue of full abstraction without them. Felleisen and Sabry show that in their model, control operators are necessary for full abstraction [SF90].

The approach in this thesis could help the search for a better denotational account inasmuch as it points out what to look for, namely self-adjointness, and what not (exponentials). One of the goals of the present approach was to give an account of continuation semantics without answers, so to speak.

Bibliography

[ACS96] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous pi-calculus. In *Seventh International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, August 1996.

[App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? *Comm. ACM*, 8:613–641, 1978.

[BCL+96] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. In *Proceedings 2nd ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes Series, December 1996.

[Bou97] Gerard Boudol. Pi-calculus in direct style. In *ACM Symposium on Principles of Programming Languages*, 1997.

[Cro93] Roy Crole. *Categories for Types*. Cambridge University Press, 1993.

[Dan94] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[DH94] Olivier Danvy and John Hatcliff. A generic account of continuation-passing styles. In *ACM Symposium on Principles of Programming Languages*, pages 458–471, 1994.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proc. ACM Symp. Principles of Programming Languages*, pages 163–173, January 1991.

[Dij68] E. W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11, 1968.

[DL92] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *ACM Conference on Lisp and Functional Programming*, pages 299–310, 1992.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991. Preliminary version in: *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, 432. Springer-Verlag (1990), 134–151.

[FF96] Matthias Felleisen and Daniel P. Friedman. *The Seasoned Schemer*. MIT Press, 1996.

[FFKD86] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings of the Symposium on Logic in Computer Science*, pages 131–141, Washington DC, June 1986. IEEE Computer Society Press.

[FH88] Anthony Field and Peter Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Fil89] Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In D. H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249. Springer-Verlag, 1989.

[Fil92] Andrzej Filinski. Linear continuations. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992.

[Fil94a] Andrzej Filinski. Recursion from iteration. *Lisp and Symbolic Computation*, 7(1), Jan 1994.

[Fil94b] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994.

[Fil96] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[Fis72] Michael J. Fischer. Lambda-calculus schemata. In *Proceedings ACM Conference on Proving Assertions about Programs*, pages 104–109, Los Cruces, 1972. SIGPLAN Notices, 7(1), January 1972.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, November 1993.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, June 1993.

[FT95] Michael Fourman and Hayo Thielecke. A proposed categorical semantics for ML modules. In David Pitt et al., editor, *Category Theory in Computer Science*, number 953 in LNCS. Springer Verlag, 1995.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 1992.

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Conference record of the 17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–58, San Francisco, CA USA, 1990.

[Has95] Masahito Hasegawa. Decomposing typed lambda calculus into a couple of categorical programming languages. In David Pitt et al., editor, *Category Theory in Computer Science*, number 953 in LNCS. Springer Verlag, 1995.

[Has97] Masahito Hasegawa. Continuation monoids. unpublished manuscript, March 1997.

[HD95] John Hatcliff and Olivier Danvy. Thunks and the lambda-calculus. Technical Report Technical report 95/3, DIKU, Computer Science Department, University of Copenhagen, February, 1995.

[HDM93] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4), October 1993.

[Hen87] Martin C. Henson. *Elements of Functional Languages.* Blackwell Scientific Publications, Oxford, 1987.

[HFW86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Journal of Computer Languages*, 11(3/4):143–153, 1986.

[Hof94] Martin Hofmann. Sound and complete axiomatizations of call-by-value control operators. *Math. Struct. in Comp. Science*, 1994.

[HS97] Martin Hofmann and Thomas Streicher. Continuation models are universal for lambda-mu-calculus. In *Proc. LICS '97*, 1997. (to appear).

[Ing61] P. Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Comm. A.C.M.*, 4(1):55–58, January 1961.

[Jac91] Bart Jacobs. *Categorical Type Theory.* PhD thesis, University of Nijmegen, 1991.

[Jac92] Bart Jacobs. Simply typed and untyped lambda calculus revisited. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*. Cambridge Univ. Press, 1992.

[KCF92] Ramma Kanneganti, Robert Cartwright, and Matthias Felleisen. SPCF: its model, calculus, and computational power. In *Proc. REX Workshop on Semantics and Concurrency, LNCS*. Springer-Verlag, 1992.

[Lai97] James Laird. Full abstraction for functional languages with control. In *Proc. LICS '97*, 1997. (to appear).

[LD93] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In *ACM Symposium on Principles of Programming Languages*, pages 124–136, 1993.

[LS86] J. Lambek and P. J. Scott. *Introduction to higher-order categorical logic.* Cambridge University Press, 1986.

[Mac71] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer Verlag, 1971.

[Mil91] Robin Milner. The polyadic π-calculus: A tutorial. LFCS Report ECS-LFCS-91-180, LFCS, University of Edinburgh, October 1991.

[Mil93] Robin Milner. Action structures for the π-calculus. Technical Report ECS–LFCS–93–264, LFCS, May 1993.

[Mil94] Robin Milner. Pi-nets: a graphical form of pi-calculus. In *European Symposium on Programming*, volume LNCS 788, pages 26–42. Springer-Verlag, 1994.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *4th LICS conference*. IEEE, 1989.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Defintion of Standard ML*. MIT Press, 1990.

[Mur91] Chet Murthy. An evaluation semantics for classical proofs. In *Proc. 11th IEEE Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1991.

[Mur92] Chet Murthy. A computational analysis of girard's translation and LC. In *IEEE Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1992.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs*, number 193 in Lecture Notes in Computer Science, pages 219–224. Springer-Verlag, 1985.

[NJ93] AT&T Bell Laboratories. *Standard ML of New Jersey — Base Environment*, 0.93 edition, February 1993.

[Ong96] C.-H. L. Ong. A semantics view of classical proofs: type-theoretic, categorical, denotational characterizations. In *Proc. 11th IEEE Annual Symposium on Logic in Computer Science*, pages 230–241. IEEE Computer Society Press, 1996.

[Par92] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings International Conference on Logic Programming and Automated Deduction*, number 624 in LNCS, pages 190–201, 1992.

[Par95] Joachim Parrow. Interaction diagrams. *Nordic Journal of Computing*, 2:407–443, 1995.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[Pow] John Power. Premonoidal categories as categories with algebraic structure. (submitted).

[PR97] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 1997. to appear.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, 1996.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[PT97] John Power and Hayo Thielecke. Environments, continuation semantics and indexed categories. In *Proceedings TACS'97*, 1997.

[RBS] J.R.B. Cockett R.F. Blute and R.A.G. Seely. Categories for computation in context and unified logic. (submitted).

[Re91] Jonathan Rees and William Clinger (editors). Revised^4 report on the algorithmic language scheme. *ACM Lisp Pointers IV*, July-September 1991.

[Ree92] Jonathan Rees. The scheme of things: The june 1992 meeting. *ACM Lisp Pointers*, 5(4):40–45, 1992.

[Rey93] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993.

[RR88] Edmund Robinson and Guiseppe Rosolini. Categories of partial map. *Information and Computation*, 79(2):95–130, 1988.

[RS94] Niels Jakob Rehof and Morten Heine Sørensen. The λ_Δ calculus. In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 516–542. Springer-Verlag, 1994.

[Sab96] Amr Sabry. Note on axiomatizing the semantics of control operators. Technical Report CIS-TR-96-03, University of Oregon, 1996.

[San94] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, may 1994.

[San95] Davide Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995.

[San97] Davide Sangiorgi. The name discipline of uniform receptiveness. In *Proc. ICALP'97*, 1997.

[Sch86] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development*. Allyn and Bacon, 1986.

[SF89] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. MIT Press, 1989.

[SF90] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: full abstraction for models of control. In M. Wand, editor, *Lisp and Functional Programming*. ACM, 1990.

[Shi96] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings 2nd ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes Series, December 1996.

[SS76] Guy Steele and Gerald Sussman. Lambda: The ultimate imperative. Technical Report AI Memo 353, MIT, March 1976.

[Ste76] Guy Steele. LAMBDA: The ultimate declarative. Technical Report AI Memo 379, MIT, November 1976.

[Ste77] Guy Steele. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. Report AI Memo 443, MIT, 1977.

[Ste78] Guy Steele. Rabbit: A compiler for Scheme. Technical Report AI TR 474, MIT, May 1978.

[SW74] Christopher Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

[SW96] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM SIGPLAN Notices*, 31(6):13–24, June 1996.

[Ten91] Robert D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, 1991.

[Thi96a] Hayo Thielecke. Continuation passing style and self-adjointness. In *Proceedings 2nd ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes Series, December 1996.

[Thi96b] Hayo Thielecke. Continuation semantics, self-adjointness and the π-calculus. Unpublished draft, March 1996.

[Thi97] Hayo Thielecke. Continuation semantics and self-adjointness. In *Proceedings MFPS XIII*, Electronic Notes in Theoretical Computer Science. Elsevier, 1997.

[Tur95] David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.