June 1992 Report No. STAN-CS-92-1426

Proceedings of the ACM SIGPLAN Workshop on Continuations CW92

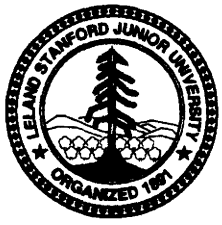
edited by

Olivier Danvy and Carolyn Talcott

Department of Computer Science

Stanford University

Stanford, California 94305



Proceedings of the ACM SIGPLAN Workshop on Continuations CW92

June 21, 1992
San Francisco, California
(between PLD192 and LFP92)

The notion of continuation is ubiquitous in many different areas of computer science, including logic, constructive mathematics, programming languages, and programming. This workshop aims at providing a forum for discussion of: new results and work in progress; work aimed at a better understanding of the nature of continuations; applications of continuations, and the relation of continuations to other areas of logic and computer science.

This technical report serves as informal proceedings for CW92. It consists of submitted manuscripts bound together according to the program order.

General chair: Olivier Danvy, danvy@cis.ksu.edu
Carolyn L. Talcott, clt@sail.stanford.edu

Contents

Anticatch

Guy L. Steele Jr.

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142

gls@think.com

One evening at POPL '92, some jolly souls were tramping down Tijeras from the hotel to the restaurant called Stephen's, chatting loudly about the theory of programming languages and generally looking like the rowdy, unruly band of marauding computer scientists that we were.

Monads were a favorite topic of discussion, thanks to Phil Wadler's tutorial on the subject that Monday morning [6], and some of us were speculating on silly and perhaps not-so-silly extensions to the examples he had given.

One such example concerned the monad of reversed state: thanks to lazy evaluation in the metalanguage, one can write an interpreter that appears to propagate state backwards through the interpreted computation (at least, relative to our intuitions about how an interpreter behaves when implemented in a. call-by-value metalanguage). The illustration was an interpreter for a language in which the computation has access to a primitive that returns the number of evaluation steps *yet to be performed* in the computation. Of course, this value must be used in such a way that the number of steps yet to be performed does not depend on the value received, lest an unresolvable mutual dependence arise. (There is probably theoretical fruit here for science fiction writers who address the paradoxes of time travel.)

On our stroll towards dinner, it occurred to me that perhaps not just derived measurement quantities, but the very values of expressions themselves could be propogated backwards through the computation by a similar technique. A few of us (including, I think, Phil Wadler, Matthias Felleisen, Olivier Danvy, and myself) discussed this more or less lightheartedly. I named the apparently necessary new language constructs anticatch and antithrow, on the grounds that whereas throw (as defined by MacLisp [4] and later Common Lisp [5]) accepts a value and transmits it to be returned by catch, I wanted antithrow to obtain and return the value that anticatch will return:

```
(anticatch (+ 3 (begin (print (antithrow)) 4)))
```

returns 7 after printing 7.

I was trying to make a dual to catch in the sense of Filinski [1, 2], in which values and continuations exchange roles and the past and future exchange roles. My intuition was that if throw gives the current value to some saved continuation, then ant **ithrow** should give a future value to the current continuation. This is what results from doing language design in one's head while walking sociably down the street. Anyway, we had a. few good laughs over it.

(Fifteen years ago, not long after catch and throw were put into MacLisp, Jon L White and I used to laugh over the idea of a. special form that could intervene between a throw and its matching catch and decide whether the catch form or the throw form should return the value. This hypothetical special form was known as bat.)

I looked at the idea for anticatch more closely on the airplane home. To simplify the semantics, I decided to use Church encoding instead of special syntactic forms (I was sure that Olivier would approve).

So I needed to invent a new function that would take another function **f** and feed it an "antithrow function" as an argument. This new function would bear the same relationship to the anticatch special form that the Scheme function call-with-current-continuation [3] bears to catch. So it looked as though it should be called call-with-function-that-returns-eventual-value.

Yuck!

But then I thought: why feed f an antithrow *function?* For such an antithrow function would take no arguments. Why not just feed **f** the eventual value itself?

So I decided to call it call-with-eventual-value, or **call/ev** for short (by analogy with the abbreviation call/cc used in a number of dialects of Scheme). Thus, as with our previous example,

```
(call/ev (lambda (v) (+ 3 (begin (print v) 4))))
```

returns 7 after printiig 7.

So the new specification is for a. function **call/ev** such that **(call/ev** f) invokes f, giving it as an argument the value that the call to **call/ev** will eventually return. And what value should the call to **call/ev** eventually return? Presumably whatever the call to f returns.

So **(call/ev f)** ≡ (f **(call/ev f)**).

How about that? call/ev ≡ **Y.**

So much for anticatch.

References

[1] Filinski, Andrzej. *Declarative Continuations and Categorical Duality*. Master's thesis. DIKU-University of Copenhagen (August 1989). DIKU Report 89/11.

[2] Filinski, Andrzej. Declarative continuations: An investigation of duality in programming language semantics. In Pitt, D. H., et al., editors, *Category Theory and Computer Science. Lecture Notes in Computer Science.* Springer-Verlag (Manchester, UK, September 1989), 224-249.

[3] *IEEE Standard for the Scheme Progmmming Language, ieee* std 11781990 edition. IEEE Computer Society (New York, 1991).

[4] Moon, David A. *MacLISP Reference Manual.* MIT Project MAC (Cambridge, Massachusetts, April 1974).

[5] Steele, Guy L., Jr., Fahlman, Scott E., Gabriel, Richard P., Moon, David A., Weinreb, Daniel L., Bobrow, Daniel G., DeMichiel, Linda G ., Keene, Sonya E., Kiczales, Gregor, Perdue, Crispin, Pitman, Kent M., Waters, Richard C., and White, Jon L. Common Lisp: *The Language (Second Edition).* Digital Press (Bedford, Massachusetts, 1990).

[6] Wadler, Philip. The essence of functional programming. In *Proc. Nineteenth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery (Albuquerque, New Mexico, January 1992), 1-14.

Recursion from Iteration

Andrzej Filinski*

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
andrzej+@cs.cmu.edu

Abstract

In a call-by-value language with first-class continuations, the usual CBV **fixpoint** combinator can be expressed in terms of a simpler iteration primitive. We give an informal derivation of this correspondence, together with a formal correctness proof. We also present a number of generalizations and possible applications.

1 Introduction

Recursive definitions in CBV functional languages have always been a bit troublesome. The usual definition of the Y-combinator doesn't work, but must be tweaked by insertion of "magical" η-redexes. In a simply-typed setting, where a recursion operator must be explicitly included as part of the language, its operational behavior is also significantly more complicated than in the CBN case. Finally, we can define recursive functions but not general recursive values. These technical problems seem to indicate that, at least in a CBV setting, recursion might more properly be viewed as a *derived, contml-specific* concept, not a ***fundamental,*** *definitional* one.

It is an elementary observation that iteration is a special case of recursion (so-called "tail recursion"). What is not so obvious is that the converse can also be true. At the implementation level, this is evident; after all, current machines only have simple loops and must keep track of recursive calls in an auxiliary data structure (typically a stack). In the following, we will see how first-class continuations can bring this correspondence up to the language level.

The best-known language with first-class continuations is of course Scheme [CR91]. However, many of the finer points and distinctions are brought out only in a statically-typed language. Fortunately, the widely available Standard ML of New Jersey compiler has an experimental first-class continuation facility (see [DHM91] for details). We will therefore use SML/NJ as the main language for examples, but give translations to Scheme where possible and appropriate.

The main difference between SML/NJ's and Scheme's first-class continuations is that the former are not represented as procedures but as values of a special type: for any type α, α `cont` is the type of o-accepting continuations. The function `callcc` provides access to the current continuation exactly like call/cc in Scheme, but applications of continuations use an explicit throw operator.

*Supported in part by NSF Grant CCR-8922109 and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

Such a presentation is somewhat more convenient in SML's polymorphic type system; it also permits a slightly more efficient implementation.

2 Fixpoints from unbounded iteration

Let us first note that adding call/cc to a simply-typed functional language without a recursion operator does not by itself make it possible to write non-terminating programs [Gri90]. We therefore need some kind of repetition construct to get started; fortunately, almost anything will do. Consider the following "endless iteration" operator:

```
fun loop f =                          | (define (loop f)
    let val rec loopf =               |   (letrec ([loopf
        fn a => loopf (f a)           |         (lambda (a) (loopf (f a)))])
    in loopf end                      |     loopf))
(* loop : ('a -> 'a) -> 'a -> 'b *)
```

(ML's fun , like Scheme's define , defines a recursive function. We will use these constructs for compactness only, resorting to an explicit val rec or letrec when we actually need recursion).

How can we use such an iterator to create a "recursor"? The key is the following observation: any function call in a CBV language is completely specified by a pair (argument value, return continuation); *we* call such a pair an *application context.* First-class continuations allow us to capture the application contexts, and thus to schedule all the calls explicitly.

Our first step is to define an application-context capturer:

```
fun switch 1 = fn x =>                | (define (switch 1) (lambda (x)
    callcc (fn q=>throw 1 (x,q))      |   (call/cc (lambda (q) (1 (cons x q))))))
(* switch : ('a * 'b cont) cont -> 'a -> 'b *)
```

(Type constructors like cont in ML bind very tightly, so 'a * 'b cont is implicitly parenthesized as 'a * ('b cont) . The 'a, usually pronounced "alpha", is ML's syntax for a generic type variable; ignore for the moment the 2 in 'b). The idea is that switch 1 looks like an ordinary function, but when applied to an 'a-typed argument with a 'b-typed return continuation, it will capture and pass these to 1:

$$k\,((switch\ l)\ v) = l\,(k, v)$$

We can now pass in this hook to our recursively-defined function and capture the full context of a recursive call:

```
fun step f => fn (v,c) =>             | (define (step f) (lambda (vc)
    callcc (fn 1 =>                   |   (call/cc (lambda (1)
      throw c                         |     ((cdr vc)
        (f (switch 1) v))             |      ((f (switch 1)) (car vc)))))))
(* step : (('2a -> '2b) -> 'c -> 'd) -> 'c * 'd cont -> 'a * '2b cont *)
```

The type if step may be a bit confusing at first sight, but the key observation is that it expresses f *as* an *application-context transformer,* mapping from a context for f to a context for f 's argument. If f never applies it argument, step does not return to the point of call.

We can now set up an iteration over contexts, intercepting all recursive calls and sending them once more through the loop; when a recursive call terminates, we pass the result to whichever continuation was waiting for it:

```
fun fix f = fn x =>                    | (define (fix f) (lambda (x)
    callcc (fn r =>                    |   (call/cc (lambda (r)
      loop (step f) (x,r))             |     ((loop (step f)) (cons x r))))))
(* fix : (('2a -> '2b) -> '2a -> '2b) -> '2a -> '2b *)
```

(The type of fix is only "weakly polymorphic" because of the corresponding restrictions on callcc [Har91]. Informally, the 2 in the type indicates that any computational effects are "protected" by at least two levels of functional abstraction. In practice this means that result computed by a recursively-defined function cannot have its type generalized (i.e., made usable at two different type instances) by a polymorphic let. Fortunately, most actual ML code does not use this generality, and there is evidence suggesting that polymorphic generalization of non-value terms is semantically questionable anyway [HL92].)

As expected from a **fixpoint** combinator, we get:

```
- fix (fn fib => fn n =>               | > ((fix (lambda (fib) (lambda (n)
        if n<2 then n                  |       (if (<n 2) n
        else fib(n-1) + fib(n-2))      |           (+ (fib (- n 1)) (fib (- n 2)))))))
      10;                              |   10)
val it = 56 : int                      | = 55
```

We use "naive Fibonacci" as our example instead of the traditional factorial function to emphasize that the recursion does not have to be linear (i.e., with at most one recursive call in the body of the defined function).

3 Correctness of the simulation

The above informal explanation of fix can be turned into a more rigorous proof, either by conversion to CPS (see section 4) or by direct-style reasoning about control operators [FH89]; let us use the latter approach here. First, we note that all of the above uses of callcc are in a restricted form in which the body of a function called by callcc is always a throw (i.e., the function called with the current continuation never returns directly to the point of call). This pattern of use for `callcc` seems quite common in programs with first-class continuations. From a Curry-Howard perspective [Gri90], it corresponds to introducing a control operator as double-negation elimination instead of the logically equivalent but less intuitively appealing Peirce's law. Operationally, it also coincides with Felleisen's rules for the C-operator on the subset of the language where the body of a C-called function is a continuation application.

Expanding the definitions above, and adopting a more compact X-syntax, we have:

$$\mathit{fix} \equiv \lambda f.\lambda x.\mathcal{C}\,(\lambda r.\mathit{loop}\,(\lambda(v,c).\mathcal{C}\,(\lambda l.c\,(f\,(\lambda x.\mathcal{C}\,(\lambda q.l\,(x,q)))\,v)))\,(x,r))$$

We can now prove that from *(loop* g) *b* = *(loop* g) (g *b),* it follows that $(\mathit{fix}\ f)\ a = f\ (\mathit{fix}\ f)\ a$:

$$\begin{aligned}
(\mathit{fix}\ f)\ a &= [\lambda x.\mathcal{C}\,(\lambda r.\mathit{loop}\,\overbrace{(\lambda(v,c).\mathcal{C}\,(\lambda l.c(f\,(\lambda x.\mathcal{C}\,(\lambda q.l\,(x,q)))\,v)))}^{F}\,(x,r))]\,a \\
&= C\,(\lambda r.\mathit{loop}\,F\,(a,r)) \\
&= C\,(\lambda r.\mathit{loop}\,F\,(F\,(a,r))) \\
&= \mathcal{C}\,(\lambda r.\mathit{loop}\,F\,([\lambda(v,c).\mathcal{C}\,(\lambda l.c(f\ \ (\lambda x.\mathcal{C}\,(\lambda q.l\,(x,q)))\,v))]\,(a,r))) \\
&= \mathcal{C}\,(\lambda r.\mathit{loop}\,F\,(\mathcal{C}\,(\lambda l.r(f\ \ (\lambda x.\mathcal{C}\,(\lambda q.l\,(x,q)))\,a))))
\end{aligned}$$

$$
\begin{aligned}
&= \mathcal{C}(\lambda r.\mathcal{C}(\lambda k.r(f\ (\lambda x.\mathcal{C}(\lambda q.\mathcal{A}(k(loop\,F(x,q)))))\,a)))\\
&= \mathcal{C}(\lambda r.\mathcal{C}(\lambda k.r(f\ (\lambda x.\mathcal{C}(\lambda q.k(loop\,F(x,q))))\,a)))\\
&= \mathcal{C}(\lambda r.r(f\ (\lambda x.\mathcal{C}(\lambda q.\mathcal{A}(loop\,F(x,q))))\,a))\\
&= \mathcal{C}(\lambda r.r(f\ (\lambda x.\mathcal{C}(\lambda q.loop\,F(x,q)))\,a))\\
&= f\ (\lambda x.\mathcal{C}(\lambda q.loop\,F(x,q)))\,a\\
&= f(\lambda x.\mathcal{C}(\lambda q.loop(\lambda(v,c).\mathcal{C}(\lambda l.c(f\,(\lambda x.\mathcal{C}(\lambda q.l(x,q)))\,v)))(x,q)))\,a\\
&= f\ (fix\,f)\,a
\end{aligned}
$$

In the above, we have used twice the identity C ($\lambda k.\mathcal{A}$ E) = C ($\lambda k.E$). This is vacuously true in a typed setting (where the **d** can never actually be executed), but also holds in general.

4 The essence of iteration

In this section, we will see in more detail why loop is a natural iteration/recursion primitive for call-by-value languages. Since the details tend to get somewhat obscured by Scheme's identification of continuations and general procedures, we will only use SML for the concrete syntax.

The characteristic equations for fixpoints and loops in a CBV language look very similar:

```
(fix f) a = f (fix f) a
(loop f ) a = (loop f) (f a)
```

However, their principal types perhaps give a better picture of their relative complexity:

```
fix  : (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)
loop : ('a -> 'a) -> ('a -> 'b)
```

Moreover, the unconstrained type variable 'b in loop shows (because of type soundness) that (loop f) a can *never* return a value. We can make this explicit by instantiating 'b to a type with no values. Let us therefore define the type void', as follows:

```
datatype void = VOID of void
fun ignore (VOID v) = ignore v (* ignore : void -> 'a *)
```

This declaration of void as an inductive type with no "base case" ensures that it has no values; ignore is the "empty function" from void to any other type, defined by a degenerate form of primitive recursion. Since ignore can never be applied to an actual value, its body does not really matter; we could equally well make it an infinite loop or raise an exception, either of which would also give it the correct type.

Now, an alternative representation of first-class continuations is as void-returning functions (see also [DHM91]). Since SML/NJ has no direct mechanism for turning such functions into continuations, we need the following idiom:

```
fun mkcont f = callcc (fn k=>ignore (f (callcc (fn c=>throw k c))))
(* mkcont : ('1a -> void) -> '1a cont *)
```

(Informally, the continuation c captures the context "pass the argument to f and do not return"; we need the other callcc to actually return c). We can now go all the way and express the fact that loop actually returns a new *continuation:*

'Not to be confused with the void found in C or some versions of ML; these correspond to SML's unit.

```
fun loop' f =
    let val rec loopf = fn a=>loopf (f a)
    in mkcont loopf end
(* loop' : ('1a -> '1a) -> '1a cont *)
```

In particular, for all f and v of appropriate types, we have:

```
throw (loop' f) v = throw (loop' f) (f v)
```

We can directly use this iterator in fix by replacing the initial call of the loop with a throw :

```
fun fix f = fn x =>
    callcc (fn r =>
      throw (loop' (step f)) (x,r))
```

From a logical point of view [Gri90], the type of *loop'* : $(a \to \alpha) \to \neg\alpha$ appears just as paradoxical as the usual *fix* : $(\alpha \to \alpha) \to \alpha$. However, to actually use the looping construct, we need some form of double-negation elimination – in the form of a call/cc operator – thereby linking non-termination to general "non-returning" behavior, i.e., escapes; the type of the **fixpoint** operator gives no such hint. In other words, loop' gives us not a recursively-defined *value,* but a recursively-defined *continuation.* And for this to be useful, the iterated function must be *non-total* (i.e., escaping), just like the usual **fixpoint** construction only makes practical sense for *non-strict* functions.

Let us finally look at the CPS conversions of loop and fix, taking o (= int for the example) as the type of final answers:

```
val rec loopc = fn f => fn a => f a (fn a'=>loopc f a') : 0
(* loopc : ('a -> ('a -> 0) -> 0) -> 'a -> 0 *)

val fixc = fn f => fn x => fn r =>
    loopc (fn (v,c)=>fn l=>f (fn x=>fn q=>l (x,q)) v c) (x,r)
(* fixc : (('a -> 'b -> o) -> 'a -> 'b -> o) -> 'a -> 'b -> o a)

fixc (fn f=>fn n=>fn c=>
        if n=0 then c 1 else f (n-1) (fn a=>c (n*a)))
     5 (fn r=>r);
(* val it = 120 : 0 *)
```

(Since loop f is a non-returning function, it doesn't need a continuation parameter). We see that loopc is in fact a **fixpoint** combinator at the CPS level (only the order of arguments is switched)! Shifting viewpoints a bit, it is the continuation semantics of CBV *iteration* that corresponds to a domain-theoretical **fixpoint,** while an explicit CBV **fixpoint** combinator f ixc just adds some administrative argument-shuffling with little inherent significance.

5 Variations

The above development was based on an infinite iteration primitive, relying on an escape to terminate the iteration. In a procedural language, this would correspond to a loop/exit construct. It is also possible to use a looping primitive with an explicit termination check (corresponding to

while or repeat-until). Here, the function to be iterated returns an explicit indication of whether another iteration should be performed, and no explicit jumps out of the loop are allowed.[2]

```
datatype ('a,'b) itres =                  ; (#f a> or (Xt b)
    AGAIN of 'a | DONE of 'b

fun repeat f =                            (define (repeat f)
    let val rec l =                         (letrec ([l (lambda (x)
        fn (DONE b) => b                          (if (car x) (cadr x)
         | (AGAIN a') = l (f a')                      (l (f (cadr x)))))])
    in fn a=>l (AGAIN a) end                  (lambda (a) (l (list #f a)))))
(* repeat : ('a -> ('a,'b) itres) -> 'a -> 'b *)
```

We still iterate over application contexts, but an additional complication is that we need an initial continuation for the first time round the loop. We make this a special case:

```
datatype 'a opt = SOME of 'a | NONE       ; <procedure> or ()

fun app (SOME k) r = throw k r            (define (app k r)
  | app NONE r = DONE r                     (if (procedure? k) (k r)
                                                (list #t r)))

fun switcha 1 = fn x=>                    (define. (switcha 1) (lambda (x)
    callcc (fn q=>                          (call/cc (lambda (q)
      throw l (AGAIN (x, SOME q)))            (l (lirt #f (cons x q)))))))

fun fix f = fn x =>                       (define (fix f) (lambda (x)
    repeat                                  ((repeat
       (fn (v,c)=>callcc (fn l =>              (lambda (vc) (call/cc (lambda (l)
          app c                                  (app (cdr vc)
              (f (switcha 1) v)))                     ((f (switcha l)) (car vc)))))))
       (x,NONE)                             (cons x '())))
```

(In Scheme, we could have used a non-continuation procedure instead of the () and app . But this would require a recursive type for the "continuation" part of the context. A similar typing problem occurs if we try to actually get an explicit representation of the stack as a list of pending contexts using the "generalized CPS" operators shift and reset [DF90].)

In procedural languages there is another kind of iteration construct, the bounded loop (usually known as for). The iteration/recursion correspondence mentioned above can be extended to this case as well, leading naturally to a notion of *bounded* recursion. In the most primitive form, we can simply define a "bounded iterator":

```
fun bloop n f =                           (define (bloop n) (lambda (f)
    let val rec l =                         (letrec ([l (lambda (n a>
        fn (0,a) => raise Bound                 (if (zero? n) (error "Bound")
         I (n,a) => l (n-l, f a)                    (l (- n 1) (f a))))])
    in fn a=>l (n,a) end                      (lambda (a) (l n a)))))
```

[2] In fact, there is a close correspondence between these two approaches: in a CBV language with first-class continuations, any function returning a sum-typed result has a unique co-curried form where one of the two cases is returned via a non-local continuation [Fil89]. In a precise sense, this is the dual concept to non-local values and currying.

We can use this to define a `bf` ix exactly as before. The immediate impression might be that a bound on the number of iterations in bloop would translate directly into a bound on the total number of recursive calls. However, a closer inspection shows that the bound actually controls the *depth* of recursion (essentially because for "parallel" recursive calls, the loop counter gets reinstantiated to the value it had when the loop continuation was captured). In particular, the Fibonacci function above will run with n as a bound. We can thus get an exponential amount of work done with what looks like a linearly-bounded primitive.

In fact, computational-complexity analysis of programs with continuations seems to be a little-explored field. For example, [DF90] presents a direct simulation of a non-deterministic finite automaton in a simple first-order functional program extended with generalized control operators. Using two levels of CPS translation, it is possible to perform collections over all paths of such a nondeterministic computation. Very speculatively, it would seem that every level of CPS adds the expressive power of an additional quantifier alternation in the polynomial hierarchy. This may or may not be related to similar-looking results about CPS transformation and logical complexity of predicate-calculus formulas [Mur91]. Also possibly relevant are the exponential-slowdown results for translation of functional programs into tail-recursive form [Kfo87].

Another way of bounding the number of iterations would be to decrement an updatable variable every time through the loop. Since the store is single-threaded, this ***will*** result in a hard bound on the number of recursive calls. More generally, the whole technique of "subverting the fixpoint" may have interesting applications – by intercepting all recursive definitions "at the root" we can express concepts like algorithmic profiling (i.e., counting recursive invocations, while remaining insensitive to compiler optimizations like in-line expansion of function bodies), engines, or preemptive thread scheduling, **etc., without explicit system support.**

6 Comparison with related work

A fair amount is known about *transforming* programs using recursion into iterative form, the so-called "flowchartability" problem. Most such work has been done in an explicitly procedural setting (e.g., [Gre75]), or for first-order recursion equations [WS73]. However, some extensions to higher-order call-by-name functional programs are reported in [Kfo87]. Interestingly, the methods in the latter work rely heavily on a notion of contexts, but the author apparently never draws any connections to continuation-passing style, let alone first-class continuations. The simulation presented above attacks a somewhat different problem: instead of considering general program transformations, we restrict ourselves to *defining* a fixpoint combinator – a construction made possible only by the additional expressive power of call/cc.

The operational derivation of fix in section 2 appears similar to "stepping" techniques used in some approaches to computational reflection [Baw88]. However, while the overall effect may be similar, the actual code (in a slightly different form) was discovered completely unexpectedly from a category-theoretical analysis of the symmetry between iteration in CBV and recursion in CBN [Fil89]. Informally, by adopting a syntax in which call/cc-like continuation abstractions look like the mirror images of ordinary X-abstractions, the abstract principle of duality can be used to expose a number of otherwise obscured symmetries involving data types, control structures and evaluation strategies.

As it turns out, a similar kind of symmetries arise for translations of either CBV or CBN λ-calculus with control operators into a system of linear *control* [Fil92]. Here, the same repetition operator corresponds to either a looping primitive in CBV or a fixpoint combinator in CBN.

7 Conclusion and Issues

The additional expressive power of first-class continuations allows us to decompose the usual CBV **fixpoint** combinator into an iterative core, whose semantics corresponds directly to a **fixpoint** combinator at the CPS level, and an administrative *wrapping* presenting a more convenient and general interface. In other words, in the presence of a call/cc-like operator, reasoning about CBV recursion can be reduced to reasoning about simple loops.

There seem to be considerable benefits from investigating first-class continuations in a typed setting. The exciting connections to classical logic [Gri90, Mur91] rely fundamentally on types, as does the author's work mentioned above. In the present investigation, the type system of SML was also a big help, ensuring that the all unbounded repetition could originate only in the loop operator itself, not from a disguised Y-like combinator in the administrative superstructure.

More generally, the equivalence of iteration and recursion gives another reason for why some form of first-class continuations should be considered a natural part of a CBV language, especially one defined by a continuation semantics. Traditionally, non-termination has been treated differently from other computational effects (i.e., the deviation of procedures from the intuitive ideal of total, set-theoretical functions). For example, a result like type soundness is often stated as: *if* the program terminates, it will do so with a result of the expected type.

In the last few years, however, there has been a shift towards a more unified view of general computation, conveniently expressible in the framework of computational monads [Mog89]. As noted by several authors, monads and CPS are very closely related [DF90, Wad92, FS92]. In this context, control operators like call/cc allow us to separate the study of non-termination (generalized to arbitrary non-returning behavior, including exceptions and non-local exits) from recursion/domain theory proper (approximations, fixpoints, etc.); in fact, what appear to be purely domain-theoretic concepts like strictness have natural generalizations in the world of control. [Fil89, Fil92]. No doubt, continuations will play an important role in any comprehensive theory of programming language semantics, far beyond what was initially imagined.

Acknowledgments

I want to thank John Reynolds for support, and Olivier Danvy, Matthias Felleisen, Dan Friedman, and the anonymous referees for their helpful comments on various versions of this paper.

References

[Baw88] Alan Bawden. Reification without evaluation. In *Proceedings of the 1966 ACM Conference on Lisp and Functional Progmmming,* pages 342-351, Snowbird, Utah, July 1988.

[CR91] William Clinger and Jonathan Rees. Revised[4] report on the algorithmic language Scheme. *Lisp Pointers,* 4(3):1–55, July 1991.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming,* pages 151-160, Nice, France, June 1990.

[DHM91] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages,* pages 163-173, Orlando, Florida, January 1991.

[FH89] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Department of Computer Science, Rice University, Houston, Texas, December 1989. To appear in *Theoretical Computer Science.*

[Fil89] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, Computer Science Department, University of Copenhagen, August 1989. DIKU Report 89/11.

[Fil92] Andrzej Filinski. Linear continuations. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Progmmming Languages,* pages 27-38, Albuquerque, New Mexico, January 1992.

[FS92] Matthias Felleisen and Amr Sabry. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming,* San Francisco, California, June 1992. (To appear).

[Gre75] Sheila Greibach. *Theory of Progmm Structures: Schemes, Semantics, Verification.* Number 36 in Lecture Notes in Computer Science. 1975.

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Progmmming Languages,* pages 47-58, San Francisco, California, January 1990.

[Har91] Robert Harper. Typing first-class- continuations in ML. In *Proceedings of the Third International Workshop on Standard ML,* Pittsburgh, PA, September 1991.

[HL92] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In *ACM SIGPLAN Workshop on Continuutions,* San Francisco, California, June 1992. (To appear).

[Kfo87] A.J. Kfoury. The translation of functional programs into tail-recursive form (part I). BUCS Tech Report 87-003, Computer Science Department, Boston University, January 1987.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science,* pages 14-23, Pacific Grove, California, June 1989. IEEE.

[Mur91] Chetan R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic* in *Computer Science,* Amsterdam, The Netherlands, July 1991.

[Wad92] Philip Wadler. The essence of functional programming (invited talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages,* pages l-14, Albuquerque, New Mexico, January 1992.

[WS73] S. A. Walker and H. R. Strong. Characterizations of flowchartable recursion. *Journal of Computer and System Sciences,* 7:404–447, *1973.*

Polymorphic Type Assignment and CPS Conversion

Robert Harper* Mark Lillibridge†
Carnegie Mellon University
Pittsburgh, PA 15213

April 10, 1992

Abstract

Meyer and Wand established that the type of a term in the simply typed X-calculus may be related in a straightforward manner to the type of its call-by-value CPS transform. This typing property may be extended to Scheme-like continuation-passing primitives, from which the soundness of these extensions follows. We study the extension of these results to the Damas-Milner polymorphic type assignment system under both the call-by-value and call-by-name interpretations. We obtain CPS transforms for the call-by-value interpretation, provided that the polymorphic let is restricted to values, and for the call-by-name interpretation with no restrictions. We prove that there is no call-by-value CPS transform for the full Damas-Milner language that validates the Meyer-Wand typing property and is equivalent to the standard call-by-value transform up to $\beta\eta$-conversion.

1 Introduction

In their study of the relationship between direct and continuation semantics for the simply typed X-calculus ($\lambda^{\rightarrow}$), Meyer and Wand note that the type of a term in $\lambda^{\rightarrow}$ may be related in a simple and natural way to the type of its call-by-value continuation passing style (CPS) transform [8]. This result may be extended to the calculus that results from extending $\lambda^{\rightarrow}$ with Scheme-like continuation-passing primitives callcc and throw ($\lambda^{\rightarrow}$ + cont) [1, 3]. Since $\lambda^{\rightarrow}$ under a call-by-value operational semantics is "type safe" in the sense of Milner [9, 2], and since the call-by-value CPS transform faithfully mimics the call-by-value semantics [12], it follows that $\lambda^{\rightarrow}$ + cont under a call-by-value operational semantics is also type safe.

In a subsequent study Duba, Harper, and MacQueen studied the addition of callcc and throw to Standard ML [10]. The extension of the Meyer-Wand transform to $\lambda^{\rightarrow}$ + cont establishes the soundness of the monomorphic fragment of the language, but the soundness of the polymorphic language with continuation-passing primitives was left open. It was subsequently proved by the authors [7] that the full polymorphic language is unsound when extended with callcc and throw. The source of this discrepancy may be traced to the interaction between the polymorphic let construct and the typing rules for callcc. Several ad hoc methods for restricting the language to recover soundness have been proposed [6, 14].

In this paper we undertake a systematic study of the interaction between continuations and polymorphism by considering the typing properties of the CPS transform for both the call-by-value and call-by-name variants of the Damas-Milner language [2] and its extension with continuation-passing primitives. We obtain suitable extensions of the Meyer-Wand theorem for the call-by-value CPS transform, provided that the polymorphic let is restricted to values, and for the call-by-name transform, under no restrictions. Finally, we

*This work was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fos Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

† Supported by a National Science Foundation Graduate Fellowship.

prove that there is no call-by-value CPS transform for the full **Damas-Milner** language that both satisfies the Meyer-Wand typing property and is equivalent to the usual transform up to $\beta\eta$-**conversion.** In particular, the standard call-by-value CPS transform fails to preserve typability.

2 Untyped Terms

The language of untyped terms is given by the following grammar:

$$\mathsf{e} \quad ::= \quad \mathsf{x} \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{let\ x\ be}\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{callcc} \mid \mathsf{throw}$$

Here x ranges over a countably infinite set of variables. We include the let construct as a primitive because it is needed in the discussion of polymorphic type assignment. callcc and throw are continuation-passing primitives whose definitions are derived from analogous constructs in Scheme [1] and Standard ML of New Jersey [3].

We consider two CPS transforms for untyped terms, corresponding to the call-by-value and call-by-name operational semantics [12]. Each CPS transform consists of a transformation $|-|$ for untyped terms and a transformation $||-||$ for untyped values. Exactly what is considered a value depends on which operational semantics is being used. Under call-by-value, variables, X-abstractions, and **constants**[1] are considered values. Under call-by-name, only X-abstractions and constants are considered values. We shall use v as a **meta**-variable for call-by-value values and w as a meta-variable for call-by-name values.

Definition 2.1 (Call-by-Value CPS Transform)

$$\begin{aligned}
|v|_{cbv} &= \lambda k.k\,||v||_{cbv} \\
|e_1\ e_2|_{cbv} &= \lambda k.|e_1|_{cbv}\,(\lambda x_1.|e_2|_{cbv}\,(\lambda x_2.x_1\,x_2\,k)) \\
|\mathsf{let\ x\ be}\ e_1\ \mathsf{in}\ e_2|_{cbv} &= \lambda k.|e_1|_{cbv}\ (\lambda x.|e_2|_{cbv}\ k)
\end{aligned}$$

$$\begin{aligned}
||x||_{cbv} &= x \\
||\lambda x.e||_{cbv} &= \lambda x.|e|_{cbv} \\
||\mathsf{callcc}||_{cbv} &= \lambda f.\lambda k.f\ k\ k \\
||\mathsf{throw}||_{cbv} &= \lambda c.\lambda k.k\,(\lambda x.\lambda l.c\,x)
\end{aligned}$$

Lemma 2.2

1. $||[v/x]v'||_{cbv} = [\,||v||_{cbv}/x\,]\,||v'||_{cbv}$.
2. $|[v/x]e|_{cbv} = [\,||v||_{cbv}/x\,]\,|e|_{cbv}$.

We shall also have need of a variant call-by-value CPS transform (cbv') defined on untyped terms satisfying the restriction that all let expressions are of the form let x be v in e. I.e., the let-bound expression is required to be a (call-by-value) value. Because of this restriction, a simpler rule can be given for the let case:

$$|\mathsf{let\ x\ be\ v\ in}\ e|_{cbv'} = \lambda k.\mathsf{let\ x\ be}\ ||v||_{cbv'}\ \mathsf{in}\ (|e|_{cbv'}\,k)$$

This simpler rule for let expressions is the only difference between the two transforms.

Lemma 2.3 *Let v and v' be values obeying the restriction on* let *expressions and e be a term obeying the restriction on* let *expressions. Then*

1. $||[v/x]v'||_{cbv'} = [\,||v||_{cbv'}/x\,]\,||v'||_{cbv'}$.
2. $|[v/x]e|_{cbv'} = [\,||v||_{cbv'}/x\,]\,|e|_{cbv'}$.

[1]Note that callcc and throw are considered to be constants.

Definition 2.4 (Call-by-Name CPS Transform)

$$
\begin{aligned}
|w|_{cbn} &= \lambda k.k\ ||w||_{cbn} \\
|x|_{cbn} &= x \\
|e_1\ e_2|_{cbn} &= \lambda k.|e_1|_{cbn}\ (\lambda x_1.x_1\ |e_2|_{cbn}\ k) \\
|\textsf{let x be } e_1 \textsf{ in } e_2|_{cbn} &= \lambda k.\textsf{let } x \textsf{ be } |e_1|_{cbn} \textsf{ in } (|e_2|_{cbn}\ k)
\end{aligned}
$$

$$
\begin{aligned}
||\lambda x.e||_{cbn} &= \lambda x.|e|_{cbn} \\
||\textsf{callcc}||_{cbn} &= \lambda f.\lambda k.f\ (\lambda f'.f'\ (X1.1\ k)\ k) \\
||\textsf{throw}||_{cbn} &= \lambda c.\lambda k.k\ (\lambda x.\lambda l.c\ (\lambda c'.x\ (\lambda x'.c'\ x')))
\end{aligned}
$$

Lemma 2.5

1. $||[e/x]w||_{cbn} = [\,|e|_{cbn}/x\,]\,||w||_{cbn}$.
2. $|\,[e/x]e'\,|_{cbn} = [\,|e|_{cbn}/x\,]\,|e'|_{cbn}$.

The correctness of these transforms may either be established by relating them to an independently-defined operational semantics (as in [12, 4]), or else taken as the definition of call-by-value and call-by-name semantics.

3 Simple Type Assignment

In this section we review Meyer and Wand's typing theorem for the call-by-value CPS transform for the simply- typed X-calculus ($\lambda^{\rightarrow}$), and present an analogous result for the call-by-name CPS transform.

Definition 3.1 ($\lambda^{\rightarrow}$ Types and Contexts)

$$\textit{types} \qquad \tau ::= b \mid \tau_1 \rightarrow \tau_2 \quad .$$

$$\textit{contexts} \quad \Gamma ::= \cdot \mid \Gamma,\ x{:}\tau$$

Here b ranges over a countable set of base types. We assume that among the base types there is a distinguished type α, which will be used in what follows to represent the "answer" type of a CPS transform.

Definition 3.2 ($\lambda^{\rightarrow}$ Typing Rules)

$$\Gamma \triangleright x : \Gamma(x) \qquad \text{(VAR)}$$

$$\frac{\Gamma,\ x{:}\tau_1 \triangleright e : \tau_2}{\Gamma \triangleright \lambda x.e : \tau_1 \rightarrow \tau_2}\ (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(ABS)}$$

$$\frac{\Gamma \triangleright e_1 : \tau_2 \rightarrow \tau \quad \Gamma \triangleright e_2 : \tau_2}{\Gamma \triangleright e_1\ e_2 : \tau} \qquad \text{(APP)}$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma,\ x{:}\tau_1 \triangleright e_2 : \tau}{\Gamma \triangleright \textsf{let } x \textsf{ be } e_1 \textsf{ in } e_2 : \tau} \qquad \text{(MONO-LET)}$$

The type system $\lambda^{\rightarrow}$ + cont is defined by adding the type espression τ cont and the following typing rules for the continuation-passing primitives:

$$\Gamma \triangleright \textsf{callcc} : (\textsf{rcont} \rightarrow \tau) \rightarrow \tau \qquad \text{(CALLCC)}$$

$$\Gamma \triangleright \textsf{throw} : \textsf{rcont} \rightarrow \tau \rightarrow \tau' \qquad \text{(THROW)}$$

Definition 3.3 (Call-by-Value Type Transform for $\lambda^{\rightarrow}$)

$$|\tau|_{cbv} = (||\tau||_{cbv} \rightarrow \alpha) \rightarrow \alpha$$

$$\begin{aligned} ||b||_{cbv} &= b \\ ||\tau_1 \rightarrow \tau_2||_{cbv} &= ||\tau_1||_{cbv} \rightarrow |\tau_2|_{cbv} \end{aligned}$$

The type transform is extended to contexts by defining $||\Gamma||_{cbv}(x) = ||\Gamma(x)||_{cbv}$ for each x $\in$ dom(Γ).

Theorem 3.4 (Meyer-Wand)

1. *If* $\lambda^{\rightarrow} \vdash \Gamma$ D $v : \tau$, *then* $\lambda^{\rightarrow} \vdash ||\Gamma||_{cbv} \triangleright ||v||_{cbv} : ||\tau||_{cbv}$.
2. *If* $\lambda^{\rightarrow}$ *I-* Γ D $e : \tau$, *then* $\lambda^{\rightarrow} \vdash ||\Gamma||_{cbv} \triangleright |e|_{cbv} : |\tau|_{cbv}$.

The'call-by-value type transform is extended to $\lambda^{\rightarrow}$ +cont by defining $||\tau\,\mathsf{cont}||_{cbv} = ||\tau||_{cbv} \rightarrow \alpha$. It is straightforward to verify that Theorem 3.4 extends to $\lambda^{\rightarrow}$ + **cont** in this way [3].

Definition 3.5 (Call-by-Name Type Transform for $\lambda^{\rightarrow}$) [2]

$$|\tau|_{cbn} = (||\tau||_{cbn} \rightarrow \alpha) \rightarrow \alpha$$

$$\begin{aligned} ||b||_{cbn} &= b \\ ||\tau_1 \rightarrow \tau_2||_{cbn} &= |\tau_1|_{cbn} \rightarrow |\tau_2|_{cbn} \end{aligned}$$

The type transform is extended to contexts by defining $|\Gamma|_{cbn}$(x) = $|\Gamma(x)|_{cbn}$ for each x $\in$ dom(Γ).

Theorem 3.6

1. *If* $\lambda^{\rightarrow}$ *t-* Γ D $w : \tau$, *then* $\lambda^{\rightarrow} \vdash |\Gamma|_{cbn} \triangleright ||w||_{cbn} : ||\tau||_{cbn}$.
2. *If* $\lambda^{\rightarrow}$ *t-* Γ D $e : \tau$, *then* $\lambda^{\rightarrow}$ *t-* $|\Gamma|_{cbn}$ D $|e|_{cbn} : |\tau|_{cbn}$.

The call-by-name CPS transform is extended to $\lambda^{\rightarrow}$ + **cont** by defining $||\tau\,\mathsf{cont}||_{cbn} = ||\tau||_{cbn} \rightarrow \alpha$, just as for call-by-value. It is straightforward to verify that Theorem 3.6 extends to $\lambda^{\rightarrow}$ + **cont** in this way.

4 Polymorphic Type Assignment

In this section we study the extension of the Meyer-Wand typing property to Damas and Milner's polymorphic type assignment system (DM).

The syntax of types and contexts in (DM) is defined by the following grammar:

Definition 4.1 (DM Types and Contexts)

$$\begin{aligned} &\textit{monotypes} \quad \tau ::= t \mid b \mid \tau_1 \rightarrow \tau_2 \\ &\textbf{\textit{polytypes}} \quad \sigma ::= \tau \mid \forall t.\sigma \\ &\textit{contexts} \quad \Gamma ::= \cdot \mid \Gamma, x{:}\sigma \end{aligned}$$

Here t ranges over a. countably infinite set of type variables. The typing rules of the Damas-Milner system extend those of $\lambda^{\rightarrow}$ as follows:

[2] The term "call-by-name type transform" is something of a misnomer since there exists a by-value CPS transform that validates the by-name typing property [5]. Nevertheless we stick with the suggestive, if somewhat misleading, terminology.

Definition 4.2 (Additional DM Typing Rules)

$$\frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall t.\sigma} \quad (t \notin FTV(\Gamma)) \qquad \text{(GEN)}$$

$$\frac{\Gamma \rhd e : \forall t.\sigma}{\Gamma \rhd e : [\tau/t]\sigma} \qquad \text{(INST)}$$

$$\frac{\Gamma \rhd e_1 : \sigma_1 \quad \Gamma, x{:}\sigma_1 \rhd e_2 : \tau_2}{\Gamma \rhd \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2 : \tau_2} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(POLY-LET)}$$

The system DM + **cont** is defined by adding the type expression τ **cont**, as before, and adding the following typing rules:

$$\Gamma \rhd \textsf{callcc} : \forall t.(t\ \textsf{cont} \to t) \to t \qquad \text{(CALLCC')}$$

$$\Gamma \rhd \textsf{throw} : \forall s.\forall t.s\ \textsf{cont} \to s \to t \qquad \text{(THROW')}$$

Let $\sigma_{\textsf{callcc}}$ and $\sigma_{\textsf{throw}}$ be the polytypes assigned to callcc and throw, respectively.

4.1 Restricted Call-by-Value

Let DM^- denote the sub-system of DM obtained by restricting let expressions so that the bound expression is a call-by-value value. The Meyer-Wand typing theorem may be extended to terms of DM^-, provided that we use the variant call-by-value CPS transform (cbv') given in Section 2.

Definition 4.3 (Call-by-Value Type Transform for DM^-)

$$\begin{aligned} |\tau|_{cbv} &= (||\tau||_{cbv} \to \alpha) \to \alpha \\ |\forall t.\sigma|_{cbv} &= \forall t.|\sigma|_{cbv} \end{aligned}$$

$$\begin{aligned} ||t||_{cbv} &= t \\ ||b||_{cbv} &= b \\ ||\tau_1 \to \tau_2||_{cbv} &= ||\tau_1||_{cbv} \to |\tau_2|_{cbv} \\ ||\forall t.\sigma||_{cbv} &= \forall t.||\sigma||_{cbv} \end{aligned}$$

This definition extends the Meyer-Wand type transform to polymorphic types. In the terminology of Reynolds [13], polymorphic instantiation is given a "trivial" interpretation in that no interesting computation can occur as a result of the specialization of a value of polymorphic type. The definition of $|\forall t.\sigma|_{cbv}$ reflects the fact that in DM^- there is no need of continuations whose domain is a polymorphic type.

Lemma 4.4

1. $||[\tau/t]\sigma||_{cbv} = [\,||\tau||_{cbv}/t\,]\,||\sigma||_{cbv}$.
2. $|[\tau/t]\sigma|_{cbv} = [\,||\tau||_{cbv}/t\,]\,|\sigma|_{cbv}$.

Theorem 4.5

1. *If* $\mathsf{DM}^- \vdash \Gamma \rhd v : \sigma$. *then* $\mathsf{DM}^- \vdash ||\Gamma||_{cbv'} \rhd ||v||_{cbv'} : ||\sigma||_{cbv}$.
2. *If* $\mathsf{DM}^- \vdash \Gamma \rhd e : \sigma$. *then* $\mathsf{DM}^- \vdash ||\Gamma||_{cbv'} \rhd |e|_{cbv'} : |\sigma|_{cbv}$.

The proof hinges on the following observations. First, the definitions of the transformations $|-|_{cbv}$ and $||-||_{cbv}$ on polytypes are such that the GEN and INST rules carry over to applications of the same rule. Specifically, if $\Gamma \rhd e : \sigma$ and t does not occur free in Γ, then t does not occur free in $||\Gamma||_{cbv'}$, and hence $||\Gamma||_{cbv'} \rhd |e|_{cbv'} : \forall t.|\sigma|_{cbv}$ is derivable by an application of GEN and the induction hypothesis. A similar argument suffices for the value transform. Uses of INST are handled similarly.

Second, the restriction on let expressions in $\mathbf{DM}^-$ combined with the use of the variant transform ensure that let's are carried over to let's, and hence that polymorphic typing is preserved. Specifically, if $\Gamma \vdash v_1 : \sigma_1$ and $\Gamma, x{:}\sigma_1 \vdash e_2 : \tau_2$ are both derivable, then by induction $||\Gamma||_{cbv'} \vdash ||v_1||_{cbv'} : ||\sigma_1||_{cbv}$ and $||\Gamma||_{cbv'}, x{:}||\sigma_1||_{cbv} \rhd |e_2|_{cbv'} : |\tau_2|_{cbv}$ are derivable, and hence $||\Gamma||_{cbv'} \vdash \lambda k.\mathsf{let}\ x\ \mathsf{be}\ ||v_1||_{cbv'}\ \mathsf{in}\ |e_2|_{cbv'}\ k : |\tau_2|_{cbv}$ is also derivable.

Theorem 4.5 extends to $\mathbf{DM}^-$ + cont by defining $||\tau\ \mathsf{cont}||_{cbv} = ||\tau||_{cbv} \to \mathrm{cr}$. We need only verify that $||\mathsf{callcc}||_{cbv'}$ and $||\mathsf{throw}||_{cbv'}$, given in Section 2, have types $||\sigma_{\mathsf{callcc}}||_{cbv}$ and $||\sigma_{\mathsf{throw}}||_{cbv}$, respectively. The soundness of $\mathbf{DM}^-$ + cont under call-by-value follows from the extended theorem. (Same proof as for the soundness of $\lambda^{\to}$ + cont under call-by-value.)

4.2 Call-by-Name

Theorem 3.6 (the Meyer-Wand-like typing theorem for call-by-name) can be extended to the unrestricted DM langauge .

Definition 4.6 (Call-by-Name Type Transform for DM)

$$
\begin{aligned}
|\tau|_{cbn} &= (||\tau||_{cbn} \to \alpha) \to \alpha \\
|\forall t.\sigma|_{cbn} &= \forall t.|\sigma|_{cbn}
\end{aligned}
$$

$$
\begin{aligned}
||t||_{cbn} &= t \\
||b||_{cbn} &= b \\
||\tau_1 \to \tau_2||_{cbn} &= |\tau_1|_{cbn} \to |\tau_2|_{cbn} \\
||\forall t.\sigma||_{cbn} &= \forall t.||\sigma||_{cbn}
\end{aligned}
$$

Lemma **4.7**

1. $||[\tau/t]\sigma||_{cbn} = [||\tau||_{cbn}/t]||\sigma||_{cbn}$.
2. $|[\tau/t]\sigma|_{cbn} = [||\tau||_{cbn}/t]|\sigma|_{cbn}$.

Theorem 4.8

1. *If* $\mathbf{DM} \vdash \Gamma \vdash w : \sigma$, *then* $\mathbf{DM} \vdash |\Gamma|_{cbn} \rhd ||w||_{cbn} : ||\sigma||_{cbn}$.
2. *If* $\mathbf{DM} \vdash \Gamma \vdash e : \sigma$, *then* $\mathbf{DM} \vdash |\Gamma|_{cbn} \rhd |e|_{cbn} : |\sigma|_{cbn}$.

The proof proceeds along similar lines to that of the call-by-value case. For example, if $\Gamma \vdash e_1 : \sigma_1$ and $\Gamma, x{:}\sigma_1 \vdash e_2 : \tau_2$ are derivable, then by induction so are $|\Gamma|_{cbn} \vdash |e_1|_{cbn} : |\sigma_1|_{cbn}$ and $|\Gamma|_{cbn}, x{:}|\sigma_1|_{cbn} \rhd |e_2|_{cbn} : |\tau_2|_{cbn}$, and hence so is $|\Gamma|_{cbn} \vdash \lambda k.\mathsf{let}\ x\ \mathsf{be}\ |e_1|_{cbn}\ \mathsf{in}\ |e_2|_{cbn}\ k : |\tau_2|_{cbn}$, as required.

Theorem 4.8 extends to DM + cont by defining $||\tau\ \mathsf{cont}||_{cbn} = ||\tau||_{cbn} \to \alpha$. We need only verify that $||\mathsf{callcc}||_{cbn}$ and $||\mathsf{throw}||_{cbn}$, given in Section 2, have types $||\sigma_{\mathsf{callcc}}||_{cbn}$ and $||\sigma_{\mathsf{throw}}||_{cbn}$, respectively. The soundness of DM + cont under call-by-name operational semantics follows from the extended theorem in a manner similar to that of the call-by-value case for $\mathbf{DM}^-$ + cont.

4.3 Unrestricted Call-by-Value

Having established suitable typing properties for the variant call-by-value transform for $\mathbf{DM}^-$ and the call-by-name transform for full DM, it is natural to consider whether there is a call-by-va.lue CPS transform for full DM that satisfies a Meyer-Wa.nd-like typing property. Since cbv' is only defined on terms with restricted let expressions, we can not simply extend Theorem 4.5 to full DM.

Let us consider attempting to extend Theorem 4.5 to full DM by using cbv instead of cbv' as the transform. Consider the induction step for the polymorphic let case. By induction we have

$$\mathbf{DM} \vdash ||\Gamma||_{cbv} \rhd |e_1|_{cbv} : |\sigma_1|_{cbv}$$

and

$$\mathbf{DM} \vdash ||\Gamma||_{cbv}, x{:}||\sigma_1||_{cbv} \rhd |e_2|_{cbv} : |\tau_2|_{cbv}.$$

We are to show that

$$\mathsf{DM} \vdash ||\Gamma||_{cbv} \triangleright \lambda k.|e_1|_{cbv}\,(\lambda x.|e_2|_{cbv}\,k) : |\tau_2|_{cbv}.$$

Since the call-by-value interpretation of let requires that e_1 be evaluated before e_2, the call-by-value CPS transform of let x be e_1 in e_2 involves a continuation whose argument may, in general, be of polymorphic type. To capture this we must change the definition of $|-|_{cbv}$ so that $|\sigma|_{cbv} = (||\sigma||_{cbv} \to \alpha) \to \alpha$. But this takes us beyond the limits of the **Damas-Milner** type system since $||\sigma||_{cbv}$ is, in general, a polytype. We therefore consider as target language the extension, DM^+, of DM, in which the distinction between monotypes and polytypes is dropped, leading to full polymorphic type assignment [11]. The decidability of type checking for DM^+ is unknown, but this is not important for our purposes. We shall rely, however, on the fact that the subject reduction property holds for P-reduction in DM^+ [11].

With these changes to the type transformation and the associated enrichment of the target type system, the induction step for general let's works. However, polymorphic generalization becomes problematic. Specifically, if DM $\vdash \Gamma \triangleright e : \sigma$ with $t \notin FTV(\Gamma)$, then by induction DM^+ t- $||\Gamma||_{cbv} \triangleright |e|_{cbv} : |\sigma|_{cbv}$, and t $\notin FTV(||\Gamma||_{cbv})$. We are to show $\mathsf{DM}^+ \vdash ||\Gamma||_{cbv} \triangleright |e| : |\forall t.\sigma|_{cbv}$, and there is no evident way to proceed. We can indeed show that $|e|$ has type $\forall t.(||\sigma||_{cbv} \to \alpha) \to \alpha$, but this is not enough. In fact we shall prove that any variant call-by-value CPS transform $|e|$ verifying the Meyer-Wand typing property for DM must not be ,&-convertible to $|e|_{cbv}$.

The argument proceeds by way of the extension of DM with continuation passing primitives. Under the call-by-value evaluation strategy, DM + **cont** is unsound. Specifically, we can find a term e such that e has a type τ, but whose value, when executed, fails to have type τ. In other words, evaluation fails to respect typing. Assuming that we have base types int and bool, and **constants**[3] 0 : int and true : bool, the following term is well-typed with type bool in DM + **cont** but evaluates under call-by-value to 0:

$$\begin{aligned} e_0 \;=\; & \text{let } f \text{ be callcc } (\lambda k.\lambda x.\mathbf{throw}\ k\ \lambda y.x) \\ & \text{in } (\lambda x.\lambda y.y)\,(f\ 0)\,(f\ \text{true}) \end{aligned}$$

Using the typing rules of DM +cont, the let-bound identifier f is assigned the type $\forall t.t \to t$, and hence may be used at types int→int and bool→bool in the body. But the binding for f grabs the continuation associated with the body of the let expression and saves it. Upon evaluation off 0, the continuation is invoked and f is effectively re-bound to a constant function returning 0. The body is re-entered, f 0 is evaluated once again (without difficulty), but then f true is evaluated, resulting in 0.

It follows that there is no call-by-value CPS transform for DM + **cont** that preserves typability. Consequently, any call-by-value CPS transform for DM must be of a somewhat different form than the usual one.

Theorem **4.9** (No Call-by-Value CPS Transform) *There is no call-by-value CPS transform* $|e|$ *for* DM *that simultaneously satisfies the following two conditions:*

1. *Equivalence:* $|e| =_{\beta\eta} |e|_{cbv}$.
2. *Typing: If* DM $\vdash \Gamma \triangleright e : \sigma$, *then* $\mathsf{DM}^+ \vdash ||\Gamma||_{cbv} \triangleright |e| : |\sigma|_{cbv}$.

Proof: Given such a transform we could form $|e_0|$ (where e_0 is given above) by regarding callcc and throw as variables of polytype $\sigma_{\mathbf{callcc}}$ and $\sigma_{\mathbf{throw}}$, respectively. By the typing property this term has type $|\mathbf{bool}|_{cbv}$, under the assumption that **callcc** and throw have types $||\sigma_{\mathbf{callcc}}||_{cbv}$ and $||\sigma_{\mathbf{throw}}||_{cbv}$, respectively. Consequently the substitution instance $e_1 = [||\mathbf{callcc}||_{cbv}, ||\mathbf{throw}||_{cbv}/\mathbf{callcc}, \text{throw}]\ |e_0|$ has type $|\mathbf{bool}|_{cbv} = (bool \to a) \to \alpha$. But the corresponding substitution instance of $|e_0|_{cbv}$ is precisely the call-by-value CPS transform of e_0, taking account of callcc and throw directly. Since &-conversion is preserved under substitution, we have by the equivalence property that e_1 is $\beta\eta$-convertible to $|e_0|_{cbv}$. Now, we know that $|e_0|_{cbv}\ \lambda x.x$ evaluates under call-by-value to 0. Consequently, this expression's $\beta\eta$ (and hence β) normal

---

"This argument can be made without constants but at the cost of increased complexity. **Constants** of base type can easily be added to any of the transforms presented in this paper by defining $||c|| = c$. c a constant. **Constants** of non-base type must be handled on a case-by-case basis.

form is 0. Therefore, we have that $e_1\ \lambda x.x$ is P-reducible to 0. But this is a violation of the subject reduction property of DM^+ [11] since $e_1\ \lambda x.x$ has type bool!

The conditions of Theorem 4.9 leave open the possibility of either finding a variant call-by-value transform that is not convertible to the standard one, or else varying the type transform in such a way that a Meyer-Wand-like typing property can be proved, or both. Any variant type transform must be such that either $||\mathsf{callcc}||_{cbv}$ or $||\mathsf{throw}||_{cbv}$ fail to have the required types under this transform so as to preclude extension to DM + cont. We know of no such variants, but have no evidence that none exist.

4.4 Related Transforms

It seems worthwhile, however, to point out that there is a variant type transform that "almost" works. This transform is defined by taking $||\forall t.\sigma|| = \forall t.|\sigma|$, and $|\sigma| = (||a|| \to \alpha) \to \alpha$. The intuition behind this choice is to regard polymorphic instantiation as a "serious" computation (in roughly the sense of Reynolds [13]). This interpretation is arguably at variance with the usual semantics of ML polymorphism since it admits primitives that have non-trivial computational effects when polymorphically instantiated. Nevertheless, we can use this type transform to extend the Meyer-Wand theorem to a variant call-by-value CPS transform for DM^- and to a variant call-by-name CPS transform for DM, provided that we restrict attention to programs of monomorphic type. It does not provide a variant call-by-value CPS transform for full DM because of the way in which polymorphic generalization is handled.

To make these observations precise, we sketch the definitions of variant CPS transforms based on this type interpretation. The main idea is to define the CPS transform by induction on typing derivations so that the effect of polymorphic generalization and instantiation can be properly handled. We give here only the two most important clauses, those governing the rules GEN and INST:

$$|\Gamma \vdash e : \forall t.\sigma| = \lambda k.k\ |e|, \text{ where}$$
$$|\Gamma \vdash e : \sigma| = |e|$$

$$|\Gamma \vdash e : [\tau/t]\sigma| = \lambda k.|e|\ (\mathrm{X}x.x\ k), \text{ where}$$
$$|\Gamma \vdash e : \forall t.\sigma| = |e|$$

This definition may be extended to the other inference rules in such a way as to implement either a call-by-name or call-by-value interpretation of application. However, the transform fails (in general) to agree with the usual (call-by-value or call-by-name) ML semantics on terms of polymorphic type. Specifically, the transformation of a GEN rule applies the current continuation to the suspended computation of e. If this continuation is not strict, then an expression that would abort in ML terminates normally after transformation into CPS. For example, consider the principal typing derivation of the term hd nil in a context assigning the obvious types to hd and nil. The resulting transform, when applied to $\lambda x.0$, will yield answer 0, despite the fact that the usual ML semantics leads to aborting in this case.

By restricting attention to programs of monomorphic type, we may obtain a correct CPS transform for DM^- (under call-by-value) and DM (under call-by-name). This is essentially because in DM^- under call-by-value there are no non-trivial polymorphic computations, and because in DM under call-by-na.me the semantics is defined by substitution. But the above argument shows that this transform is incorrect, for DM under call-by-value. Specifically, it fails to correctly implement the usual ML semantics for expressions such as let x be hd nil in 0 (which, under the above transformation yields result 0 rather than aborting).

5 Conclusion

The Meyer-Wand typing theorem for the call-by-value CPS transform for the simply-typed λ-calculus establishes a simple and natural relationship between the type of a term and the type of its call-by-value CPS transform. Meyer and Wand exploited this relationship in their proof of the equivalence of the direct and continuation semantics of $\lambda^{\to}$ [8]. A minor extension of this result may be used to establish the soundness of typing for $\lambda^{\to}$ + cont, the extension of $\lambda^{\to}$ with continuation-passing primitives [3], under call-by-value.

In this paper we have presented a. systematic study of the extension of the Meyer-Wand theorem to the Damas-Milner system of polymorphic type assignment. Our main positive results are the extension of the

Meyer-Wand theorem to the call-by-value interpretation of a restricted form of polymorphism, and to the call-by-name interpretation of the unrestricted language. These results have as a consequence the soundness (in the sense of Damas and Milner [2]) of these programming languages. We have also argued that there is no "natural" call-by-value CPS transform for the unrestricted language, but this leaves open the possibility of finding a transformation that is radically different in character from the usual one.

Our investigation makes clear that there is a fundamental tension between implicit polymorphism and the by-value interpretation of let. In particular, we are able to provide a CPS transform for the full Damas-Milner language that extends to continuation-passing primitives, but which is "not quite" equivalent to the usual call-by-value semantics. This suggests that a language in which polymorphic generalization and instantiation are semantically significant would be well-behaved, and might be a suitable alternative to ML-style implicit polymorphism. We plan to report on this subject in a future paper.

6 Acknowledgments

We are grateful to Olivier Danvy, Tim Griffin, Mark Leone, and the referees for their helpful comments on earlier drafts of this paper.

References

[1] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for higher-level semantic algebra. In Maurice Nivat and John C. Reynolds, editors, Algebraic *Methods in Semantics,* pages 237-250. Cambridge University Press, Cambridge, 1985.

[2] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Progmmming Languages,* pages 207-212, 1982.

[3] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth A CM Symposium on Principles of Progmmming Languages,* January 1991.

[4] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science,* 52(3):205–237, 1987.

[5] Timothy Griffin. Private communication., January 1992.

[6] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. Revised and expanded version of [3], in preparation.

[7] Robert Harper and Mark Lillibridge. Announcement on the types electronic forum., July 1991.

[8] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda calculi (summary). In Rohit Parikh, editor, *Logics of Programs,* volume *224* of *Lecture Notes in Computer Science,* pages 219-224. Springer-Verlag, 1985.

[9] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System sciences,* 17:348–375, 1978.

[10] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[11] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In *1986 Symposium. on LISP and Functional Programming,* pages 308-319, August 1986.

[12] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer. Science,* 1:125–159, 1975.

[13] John C. Reynolds. Definitiona. interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference,* pages 717–740, Boston, August 1972. ACM.

[14] Andrew K. Wright and Matthias %elleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, July 1991.

Continuations and simple types: a strong normalization result

Franco Barbanera *Stefano Berardi*
Universitá di Torino
Dipartimento di Informatica
Corso Svizzera, 185 10149 Torino (Italy)
barba,stefano@di.unito.it

Abstract

In the paper we study the termination problem for a typed X-calculus with continuations. We do not bound ourselves to study a particular reduction strategy, like call-by-value or call-by-name. Reductions may be applied to any part of any term in any order.

Our main result is that every reduction sequence in the system terminates.

1 Introduction

Recentely, in the computer science community, efforts have been made in order to understand and investigate functional languages enriched with so called *control operators*. One esample of such control operators the Call/cc of the programming language SCHEME [3]. An operator like that allows us to have, even in a. functional enviroment, a sort of goto-like facility which enable the programmers to write, in many cases, shorter and more coincise programs. This is of course a useful feature, but one could now wonder if such a feature, when embedded in a functional programming language, does not destroy any of its good properties. In particular, those peculiar properties of *pure* functiona. languages which have been the main motivations of their development , namely espressiveness, possibility of developing programs satisfying their specifications and so on. This risk is a serious one. However, the benefits we can gain from control operakors axe definitely worth a study and investigation. What one needs is to frame control operakors for functional languages in a. correct and clear theoretical setting. Such a theoretical setting is the necessary basis for a deep understanding of control operators and would enable one to define precise and "harmless" methodology for their use. In fact, that has been the main motivation which led people like Felleisen to investigate pure calculi in which to isolate and study the properties of control operators. A possible *pure* control language is the pure lambda calculus enriched with a particular operator $\mathcal{C}$ ($\lambda_{\mathcal{C}}$) [4],[5]. $\mathcal{C}$ which can be considered as an abstraction of the actual control operators. To study this language it is possible to give a machine-like operational semantics or to define a calculus extending the notion of β-reduction of the X-calculus. Of course, for the purpose of theoretical investigation it is better not to restrict oneself just to machine-like operational semantics, since forma.l calculi allow precise mathematical reasoning about programs. In particular they allow reasoning abou t programs equivalences under various possible evaluators. Such calculi and their equational theory have been and axe still widely investigated [4][5].

More recentely, the in terest for control calculi has increased even more because of the st rong connection which has been unveiled with classical logics. Recent results ([6],[7],[8]) have shown

that there exists *a* precise *proofs-as-programs* correspondence between classical proofs and control functional languages. More precisely, Fellaisen's control operator C can be consistently interpreted as the "computational content" of the classical rule of double negation elimination. This correspondence with classical logics could be in the future a key tool for the design of environments for developing programs in functional languages with control operators (and maybe also imperative languages), in the same way the correspondence with constructive logics has been (and is still) of much help in the field of pure functional languages. Of course there is still a lot to do. The correspondence with the logics, for instance, makes necessary to investigate more closely *typed* version of control calculi. Besides, the correspondence results have been obtained just for control calculi with particular reduction strategies. Each time one wishes to use a new strategy one has to check for it all the properties we whishes the calculus to have.

The present paper tackles some of these problems. We consider a typed control calculus $\lambda_{\mathcal{C}\tau-}$ in the style of [6],[7],[8]. Of this calculus we prove the property of *strong normalizability* without sticking to any particular reduction strategies. Such a property has never been considered as relevant for such calculi, even because it is widely believed that control operators have a sense only in languages with precise reduction strategies. This is not our opinion. First of all because, given a calculus, it is a properties worth being investigated by itself. Besides, strong normalizability for reductions performed in whatever order allows one to settle the question of termination of any possible reduction strategy once and for all.

There is another relevant argument which leads to investigate a continuation calculus in which no fixed reduction strategy is given. Once one considers terms in the typed $\lambda\mathcal{C}$ as classical proofs, the reduction rules of the calculus can be viewed simply as reduction on proofs, enabling to extract, for instance, witnesses from existentially quantified classical proofs, in the style of [1],[2] and [9]. By not forcing a particular reduction strategy we get a calculus which is not Church-Rosser. This means that different reductions strategies allow us to get different " answers" (witnesses) from a given term (proof) which we can now look at as a non-deterministic program. If we restricted ourselves to a fixed reduction strategy, we would choose only one answer, in a. rather arbitrary way. In a sense, if we forced it to be deterministic we would "mutilate" the algorithm implicitly expressed by a classical proof.

The proof of strong normalization use a non trivia.1 variant of Girard's method of candidates of reducibility.

In Section 2 we shall describe the control calculus $\lambda_{\mathcal{C}\tau-}$, while whole Section 3 is occupied by the strong normalization proof.

2 The system AC_T--

In this section we shall describe the *typed* system $\lambda_{\mathcal{C}\tau-}$ and the set or reduction rules on its terms. The terms of $\lambda_{\mathcal{C}\tau-}$ are typed lambda, terms enriched with the control operator $\mathcal{C}$. The typing for the terms will follows the one proposed in [6] [7]which is essentially a. typing for *classical* proofs. This means that our types are formulas, and terms are anything but *linearized* classical proofs. For motivations we shall make clear in the following, we restrict the types of $\lambda_{\mathcal{C}\tau-}$ to a strict subset of a.11 the possible logical formulas (the superscript • ?-'? on the τ in the name $\lambda_{\mathcal{C}\tau-}$, espressing in turn the " typefulness" of the system, is to recall that we do not. consider all the possible types).

The types of our system are a. subset of the simple types á la Curry, i.e. of the types built out of atomic types a, b, c,. . . and rising the con nectives $\rightarrow$ ("im plication") and $\perp$ ("falsehood"). The

negation in our system is defined as usual, by

$$\neg A =_{Def} A \to \bot.$$

We restrict the types á la Curry by forbidding types to have ***strict*** subtypes of the form $\neg\neg A$. We also forbid $\bot$ to occur on the righthand side of $\to$ (like in $\bot \to A$).
Hence $\neg\neg A$ and $\neg(B \to \neg A)$ $(A,\ B \neq \bot)$ are types of our system, while $A \to \neg\neg B$ and $A \to (\bot \to B)$ are not.
This restrictions on types is indeed no restriction for the classical logic associated to the calculus: in classical logic $\neg\neg A$ can always be replaced by A. Moreover types of the form $\bot \to A$ are of no real use, since from $\bot$ we can deduce any A.

The formal definition of the types of $\lambda_{C\tau-}$ runs as follows:

Definition **2.1** *The sets* of Positive types *(P),* Negative types *(N),* PosNeg types *(PN) and* Double negated types (NN) *are defined by the following grammars, where a denotes the set of type constants*

$P ::= a|P \to P|P \to \neg P|\neg P \to P|\neg P \to \neg P$
$N := \neg P$
$PN := P|N$
$NN := \neg\neg P.$
The set of Types *(T) of* $\lambda_{C\tau-}$ *is the union* of *the sets defined above, i.e. it is defined by*

$$T := \bot|P|N|NN$$

Then positive types are those which are not negations; $\bot$ can occur in a type only in subtypes of the form $\neg A$ and a double negated type can occur alone, but not inside other types.
In the following, positive types will be denoted by P, P', P",. . . . Types and PosNeg types will be denoted by T, A, B, C,. . . (then if A is a. PosNeg type, $\neg A$ will denote a correct type, maybe double negated, while $\neg\neg A$ instead may be out of the set of correct types, if A is itself negated).

For each type T, we suppose to have infinitely many variables labelled with T:

$$\mathrm{Var}_T =_{Def} x_0^T, x_1^T, x_2^T, \ldots$$

We shall not use the label T when it will be clear from the contest.

We define now a set of "pseudoterms" and a set of typing rules. The set of terms of system $\lambda_{C\tau-}$ will be the pseudoterms having a. correct type. The pseudoterms are built out of variables, using abstraction, application and the operators $\mathcal{C}$ (continuation operator) and $\mathcal{A}$ (abort operator). We shall assume each occurrence of the operator A to have a type label $T \neq \bot$ $(\mathcal{A}_T)$ which we shall not show when unnecessary.

Definition **2.2** *The set* of Pseudoterms of $\lambda_{C\tau-}$ *is defined by the following grammar :*

$$M ::= x^T|\lambda x.M|(MM)|\mathcal{C}M|\mathcal{A}_T M$$

Definition 2.3 (Typing rules) *Let A, B be* ***PosNeg*** *types, P a positive type, and* ***M, N pseudoterms.***

$$var)\ x^A : A$$

$$\to\!I)\ \frac{M : B}{\lambda x^A.M : A \to B} \qquad \to E)\ \frac{M : A \to B \quad N : A}{MN : B}$$

$$\neg I)\ \frac{M : \ \bot}{\lambda x^A.M : \neg A} \qquad \neg E)\ \frac{M : \neg A \quad N : A}{MN : \bot}$$

$$\neg\neg E)\ \frac{M : \neg\neg P}{\mathcal{C}M : P} \qquad \mathcal{A})\ \frac{M : \bot}{\mathcal{A}_T M : T}$$

We call then term *a pseudo term having a correct type.*

It is not difficult to see that the type of a term is unique (because of the type labels on variables) and may be computed.
We shall denote by $\mathbf{Term}_T$ the set of terms having type T.
A term of the form CM will be called *a continuation.* One of the form AM an *abort.*

We introduce now reductions on terms, and define strong normalization for them. Reductions $\mathcal{C}_R$ and $\mathcal{C}_L$ are the typed version of Felleisen's reductions [4][5]. To deal with the case when the redexes $(\mathcal{C}M)N$ or $M(CN)$ have a negative type we introduce rules $\mathcal{C}'_R$ and $\mathcal{C}'_L$ that, instead of moving the continuation outside, make it disappear. This is possible because a triple negation is intuitionistically equivalent to a negation. The necessity of having these latter rules depends on the restriction on types we imposed to the system. Rule $\mathcal{C}''_R$ is an instance of the general rule $E[\mathcal{C}M] \to_1 M(\lambda x.E[x])$ for continuations and is introduced in order to deal with the case of the elimination of negation.

Definition *2.4* (Reduction rules)

$\beta)$	$(\lambda x.M)N$	$\to_1$	$M[N/x]$	
$\mathcal{C}_L)$	$(\mathcal{C}M)N$	$\to_1$	$\mathcal{C}\lambda k.M(\lambda f.k(fN))$	$(^1)$
$\mathcal{C}'_L)$	$(\mathcal{C}M)N$	$\to_1$	$\lambda p.M(\lambda f.(f\ N)p)$	$(^2)$
$\mathcal{C}_R)$	$M(\mathcal{C}N)$	$\to_1$	$\mathcal{C}\lambda k.N(\lambda a.k(Ma))$	$(^3)$
$\mathcal{C}'_R)$	$M(\mathcal{C}N)$	$\to_1$	$\lambda p.N(\lambda a.(Ma)p)$	$(^4)$
$\mathcal{C}''_R)$	$M(CN)$	$\to_1$	$N(\lambda a.(Ma))$	δ
$\mathcal{A})$	$E[\mathcal{A}M]$	$\to_1$	M	δ

Provisos:
$(^1)$ *M has to have type of the form $\neg\neg(A \to P)$*
$(^2)$ *M has to have type of the form $\neg\neg(A \to \neg P)$*
$(^3)$ *M has to have type of the form $A \to P$*
$(^4)$ *M has to have type of the form $A \to \neg P$*
$(^5)$ *M has to have type of the form $\neg P$*
$(^6)$ *$E[-]$ is a* ***context*** *$\neq [-]$ with type $\bot$ and $FV(M) \subseteq FV(E[\mathcal{A}M])$*

$\to$ will denote the reflexive and transitive closure of $\to_1$.

In the reductions defined above we have not put the type decorations for sake of **semplicity.** We give below the reduction rules with all the type decorations.

β) $((\lambda x^A.M^B)^{A\to B}N^A)^B \to_1 (M[N/x])^B$

C_L) $((\mathcal{C}(M)^{\neg\neg(A\to P)})^{A\to P}N^A)^P \to_1$
$(\mathcal{C}(\lambda k^{\neg P}.(M(\lambda f^{A\to P}.(k(fN))^{\perp})^{\neg(A\to B)})^{\perp})^{\neg\neg P})^P$

C'_L) $((\mathcal{C}(M)^{\neg\neg(A\to\neg P)})^{A\to\neg P}N^A)^{\neg P} \to_1$
$(\lambda p^P.(M(\lambda f^{A\to\neg P}.((fN)^{\neg P}p)^{\perp})^{\neg(A\to\neg P)})^{\perp})^{\neg P}$

C_R) $(M^{A\to P}(\mathcal{C}(N)^{\neg\neg A})^A)^P \to_1$
$(\mathcal{C}(\lambda k^{\neg P}.(N(\lambda a^A.(k(Ma)^P)^{\perp})^{\neg A})^{\perp})^{\neg\neg P})^P$

C'_R) $(M^{A\to\neg P}(\mathcal{C}(N)^{\neg\neg A})^A)^{\neg P} \to_1$
$(\lambda p^P.(N(\lambda a^A.((Ma)^{\neg P}p)^{\perp})^{\neg A})^{\perp})^{\neg P}$

C''_R) $(M^{\neg A}(\mathcal{C}(N)^{\neg\neg A})^A)^{\perp} \to_1 (N(\lambda a^A.(Ma)^{\perp})^{\neg A})^{\perp}$

$\mathcal{A}$) $(E[(\mathcal{A}M^{\perp})^T])^{\perp} \to_1 M^{\perp}$

Definition 2.5 *Let n be an* integer, M *a term and T a type.*
i) n is 'a bound for M *if the reduction tree of* M *has a finite height* $\leq n$.
ii) M *strongly normalizes if it has a bound.*
iii) $SN_T = \{ M \in Term_T \mid M$ *strongly normalizes)*

Then a term strongly normalizes iff its reduction tree is finite.
This definition has to be preferred to the usual one, i.e. "each reduction sequence from M is finite", because the latter is intuitionistically weaker than the former (classically, they are equivalent through **König** Lemma).

In the nest section we shall give the proof of strong normalization for terms of $\lambda_{\mathcal{C}_T-}$.

3 Strong normalization for $\lambda_{\mathcal{C}_T-}$

Our proof method of strong normalization is essentially a non-trivial modification of Girard's candidates. We sketch briefly now why, even with the restriction on types, Girard's method is not applicable direcly as it is, and what are the modifications we made to it.
Girard's method, following Tait, is based on a notion of "computability". Computable terms strongly normalizes. Thus the goal to prove strong normalization becomes to show that each term is computable, a. thing that is not difficult to prove by induction on the term. To define a notion of computability for continuations, instead, is not easy. The first attempt which naturally would come in mind, in order to define a notion of computability, would be the following :

1. a. variable is computable outright.
2. $\lambda x.M$ is computable if, for all computable terms N with the same type of x, $M[N/x]$ is computable.
3. C(M) is computable if M is computable.
4. MN is computable if it strongly normalizes, and all its reducts which are not applications are computable (i.e.? they satisfy either 1., or 2., or 3.).

This definition is uncorrect as stated. While 2. is a definition by induction on the type of the term, 3. forces a circularity. By 3., the computable terms C(M) of type A are defined from the coin pu table terms of type $\neg\neg A$. The latters, by 2., are defined form the computable terms of type $\neg A$, and by 2. again, from the computable terms of type A.

What we have done is to break this cycle by stratifying the above definition over an ordinal parameter, i.e. by considering it as a general inductive definition and using this ordinal induction during the proof.
We add terms of the form CA4 to the interpretation of A by steps. In the first step we put in the interpretation of A only the terms which are not continuations. Then in step $\alpha + 1$ we considers the terms produced in step α, and we add to A all the terms CM such that M introduced in $\neg\neg A$ at the α^{th} step. For Tarski theorem it is impossible to go on indefinitely in adding terms to the interpretation of A; we have to stop *at most* at step ω_1, the frst uncountable ordinal.

A formal definition of our notion of computability will be given in the next subsection.

3.1 Stratified Girard candidates for $\lambda_{C\tau-}$

In this section we define a notion of candidates for the language, and we associate a candidate to each type.

Definition 3.1 *A set X of terms is a candidate for a type T iff the following conditions hold :*

Cand0) $X \supseteq Var_T$

Candl) $SN_T \supseteq X$

Cand2) *X is closed by reductions*

Cand3) for every $MN \in Term_T$: *if* $(\forall Q \in Term_T)(MN \to_1 Q \Rightarrow Q \in X)$ *then* $MN \in X$.

Lemma 3.1 $SN_\perp$ *is a candidate for 1.*

Proof.
Straightforward. • I

In order to associate a particular candidate to each type, we first define some operators on sets of terms : Lambda, Lambda$^\neg$, Not, Ap, Clos, Cont, Abort, having the following functionality

Lambda :	$\mathcal{P}(Term_A) \to \mathcal{P}(Term_B) \to \mathcal{P}(Term_{A\to B})$
Lambda', not :	$\mathcal{P}(Term_A) \to \mathcal{P}(Term_A)$
Ap, Clos, Cont :	$\mathcal{P}(Term_A) \to \mathcal{P}(Term_A)$
Abort.4 :	$\mathcal{P}(Term_A)$

Definition 3.2 *Let A, B be types, $X \in \mathcal{P}(Term_A)$, $Y \in \mathcal{P}(Term_B)$, $Z \in \mathcal{P}(Term_{\neg\neg P})$. We define :*

$$
\begin{array}{lll}
Abort_A & =_{Def} & \{\mathcal{A}_A M \in Term_A \mid M \in SN_\perp\} \\
Lambda(X,Y) & =_{Def} & \{\lambda x.M \in Term_{A\to B} \mid (\forall Q \in X)(M[Q/x] \in Y)\} \\
Lambda^\neg(X) & =_{Def} & \{\lambda x.M \in Term_{\neg A} \mid (\forall Q \in X)(M[Q/x] \in SN_\perp)\} \\
Ap(X) & =_{Def} & \{MN \in SN_A \mid \\
 & & (\forall Q \in Term_A)(Q \text{ not } application, MN \to Q) \Rightarrow Q \in X\} \\
Clos(X) & =_{Def} & X \cup Ap(X) \\
Not(X) & =_{Def} & Clos(Var_{\neg A} \cup Abort_{\neg A} \cup Lambda^\neg(X)) \\
Cont(Z) & =_{Def} & \{\mathcal{C}M \in Term_P \mid M \in Z\}
\end{array}
$$

It is possible to see that Lambda and Lambda' express the constructive meaning of lambda abstraction, while Ap says that the constructive meaning of a term MN depends on the constructive meaning of its reducts. The use of the operator Clos is to close a set of terms X under Ap. The operator Not translates the constructive meaning of the negation. Cont expresses that the constructive meaning of a term CM is nothing but the constructive meaning of M.

It is not difficult to see that the operator Lambda' is decreasing w.r.t. the set-theoretical inclusion order; Ap and Continuation are instead increasing. ¿From the observations above it easily descends that Not is decreasing and NotoNot is increasing.

We are now ready to define, for each type T, a candidate $[T]$ associated to it. If T is positive, we shall define $[T]$ as the ω_1 limit of an increasing chain $[T]_\alpha$ of subsets of Term_T, where a denote an ordinal and ω_1 is the first uncountable ordinal. For non positive types the associated candidate will be defined using the definition of candidate for positive types and the operator Not.

Definition 3.3 *(i)* $[1] =_{Def} SN_\perp$.
(ii) Let T be a type. We define $[T]$ and $[T], \in \mathcal{P}(Term_T)$, for each ordinal α, as follows:

. $\alpha = 0$

1. *If P is an atomic type $a \neq \perp$:*
   $[P]_0 =_{Def} Clos(Var_P \cup Abortp)$
2. *If P is $A \to B$, assume to have already defined $[P']$ for all subtypes P' of P :*
   $[P]_0 =_{Def} Clos(Var_P \cup Abort_P \cup Lambda([A], [B])$
3. *If $T = \neg P$:*
   $[P]_0 =_{Def} Not([P]_0)$

- $\alpha = \gamma + 1$

1. *If T is positive :*
   $[T]_{\gamma+1} =_{Def} Clos([T]_\gamma \cup Cont([\neg\neg T]_\gamma))$
2. *If $T = \neg A$:*
   $[T]_{\gamma+1} =_{Def} Not([A]_{\gamma+1})$

- *$a \ast$ a limit ordinal β :*

1. *If T is positive :*
   $[T]_\beta =_{Def} \bigcup_{\gamma<\beta}[T]_\alpha$
2. *If $T = \neg A$:*
   $[T]_\beta =_{Def} Not([A]_\beta)$

[T] is now defined as follows :
$[T] =_{Def} [T]_{\omega_1}$

If P is a positive type it is easy to check that the chain $[P]_\alpha$ is increasing because $[P]_{\alpha+1} = \mathrm{Clos}([P]_\alpha \cup \ldots) \supseteq [P]_\alpha$. Therefore, by Tarski's Fixed Point Theorem we get $[P] = [P]_{\omega_1} = [P]_{\omega_1+1}$. Thus, by putting $\alpha = \omega_1$ in the definition of $[P]_{\alpha+1}$ we have that:

$$[P] = \mathrm{Clos}([P] \cup \mathrm{Cont}([\neg\neg P]))$$

If $T = \neg A$, by the same argument we have instead $[T] = \mathrm{Not}([A])$.

Definition 3.4 *Let M be a term and A its type.*
Following Tait, we shall call **M** computable ***iff*** $M \in [A]$.

Later we shall prove that every term is computable. It will follow, by Candl, that every term strongly normalizes.

3.1.1 Compound candidates

In this subsection we shall check that *[T],* defined previously, is a candidate for any type *T.*

In the following Lemmas 3.2-3.5 we shall prove relevant properties of the operators we introduced. Then we shall be able to prove (in Lemma 3.6) that for each type T and ordinal α, the set $[T]_\alpha$ (in particular *[T])* is a candidate.

Lemma 3.2 *Let T be type and* $X \in \mathcal{P}(Term_T)$. *Then:*
X *satisfies* ***Cand0, Candl, Cand2*** $\Rightarrow$ ***Clos(X)*** *is a candidate.*

Proof:
We check separately **Cand0,**. . . , Cand3 for $\mathbf{Clos}(X)$.
Recall that $\mathbf{Clos}(X)$ = X $\cup$ $\mathbf{Ap}(X)$.

Cand0) $\mathbf{Clos}(X) \supseteq X \supseteq \mathbf{Var}_T$.

Candl) $SN_T \supseteq \mathbf{Clos}(X)$ by definition of Ap and $SN_T \supseteq$ X.

Cand2) Assume $M \in \mathrm{Clos}(X)$ and $M \to$ N in order to prove N $\in \mathrm{Clos}(X)$. Then either $M \in X$, or $M \in \mathrm{Ap}(X)$. In the first case, we apply Cand2 on X to $M \to$ N to deduce $N \in X$. Thus, N $\in \mathrm{Clos}(X)$ because $\mathrm{Clos}(X) \supseteq$ X. In the second case, by definition of Ap we know that $M = M_1M_2 \in SN_T$, and that

$$(\forall Q \in \mathrm{Term}_A)\ (\mathrm{Q\ not\ application\ and}\ M_1M_2 \to Q) \Rightarrow Q \in X \qquad (1)$$

Suppose now N not to be an application. Then from $M_1\ M_2 \to$ N we deduce N $\in X$ by 1. We are so reduced to the first case. If N is instead an application, then, since (N $\to$ Q) $\Rightarrow$ ($M_1M_2 \to$ Q), from 1 we conclude:

$$(\forall Q \in \mathrm{Term}_A)(Q\ \mathrm{not\ application\ and}\ N \to \mathrm{Q}) \Rightarrow \mathrm{Q} \in X) \qquad (2)$$

Therefore, N $\in \mathrm{Ap}(X)$ by definition of Ap. Thus, $N \in \mathrm{Clos}(X)$ because $\mathrm{Clos}(X) \supseteq \mathrm{Ap}(X)$.

Cand3) Assume:

$$(\forall Q \in \mathrm{Term}_A)(MN \to_1 \mathrm{Q}) \Rightarrow \mathrm{Q} \in \mathrm{Clos}(X) \qquad (3)$$

in order to prove $MN \in \mathrm{Clos}(X)$. It is indeed enough to prove MN $\in \mathrm{Ap}(X)$. There are finitely many Q's such that MN $\to_1 Q$; say, $Q_1, \ldots, Q_n$. Since each $Q_i \in \mathrm{Clos}(X)$, and $\mathrm{Clos}(X)$ satisfies Candl), then each Q_i has a bound n_i. Therefore, $\max_i\{n_i + 1\}$ is a. bound for MN, and $MN \in SN_A$. To prove $MN \in \mathrm{Ap}(X)$, there is still left to check:

$$(\forall Q \in \mathrm{Term}_A)\,(Q\ \mathrm{not\ application\ and}\ MN \to \mathrm{Q}) \Rightarrow \mathrm{Q} \in X \qquad (4)$$

To prove 4, assume Q is not an application, and $MN \to Q$. Then $\mathrm{Q} \neq MN$. It. follows that. for some Q' we have MN $\to_1$ Q' $\to$ Q. By the assumption 3, Q' $\in \mathrm{Clos}(X)$. Then Q $\in \mathrm{Clos}(X)$ follows by Q' $\to Q$, because $\mathrm{Clos}(X)$ satisfies Cand2. Since Q is not an application, t hen Q $\notin \mathrm{Ap}(X)$, and therefore Q $\in X$. • I

Lemma *3.3 Let A, B be positive or negative types, C any type* $\neq I$, $X \in \mathcal{P}(Term_A)$, $Y \in \mathcal{P}(Term_B), Z, Z' \in (Term_C)$.

(i) X satisfies **Cand0**, *Y satisfies Candl, Cand2* $\Rightarrow$ **Lambda(X,Y)** *satisfies Candl,* **Cand2.**

(ii) X satisfies **Cand0** $\Rightarrow$ *Lambda'(X) satisfies Candl,* **Cand2.**

(iii) (Z, Z' satisfy Candl, Cand2) and (Z or Z' satisfy also **Cand0**) $\Rightarrow$ $Z \cup Z'$ *satisfies* **Cand0,** *Candl and Cand2.*

(iv) $\mathbf{Var_C}$ *satisfies* **Cand0,** *Candl and* **Cand2.**

(v) $\mathbf{Var_C \cup Abort_C}$ *satisfies* **Cand0,** *Candl and Cand2.*

Proof.
i) Let $\lambda x.M \in$ Lambda(X, Y). We check Candl and Cand2 separately.

Candl) We have to prove that $\lambda x.M \in SN_{A\to B}$. By Cand0, $x \in$ X; by definition of Lambda(X,Y), it follows $M \in$ Y; by Candl, we deduce $M \in SN_B$. Since $\lambda x.M$ has not type I, it is not an $\mathcal{A}$-redex, and each reduction out of $\lambda x.M$ is indeed a reduction on M and any bound for M is a bound for $\lambda x.M$ as well. We conclude that $\lambda x.M \in SN_{A\to B}$.

Cand2) Assume $\lambda x.M \to$ N in order to prove N $\in$ Lambda(X, Y). Since $\lambda x.M$ is not an d-recles, each reduction out of $\lambda x.M$ is indeed a reduction on M; then $N = \lambda x.Q$ and $M \to Q$. Therefore, it is enough to check that, for every S $\in X$, $Q[S/x] \in$ Y. By $\lambda x.M \in$ Lambda(X, Y) it follows that $M[S/x] \in$ Y. Since $M[S/x] \to Q[S/x]$, applying Cand2 for Y we conclude that $Q[S/x] \in$ Y. Thus, $N \in$ Lambda(X, Y).

ii) Analogous to point i). We use the fact that $SN_\perp$ is a candidate.
iii) If Z $\supseteq$ Var_C, then Z $\cup$ $Z' \supseteq \mathrm{Var}_C$. If S $N_C \supseteq Z$, Z', then S $\mathrm{N}_C \supseteq Z \cup$ Z'. Assume now Z and Z' to be closed by reduction, and that $M \in Z \cup$ Z', $M \to$ N. If $M \in Z$ then N $\in Z$; if $M \in Z'$ then N $\in$ Z'. In both cases, N $\in$ Z $\cup$ Z'.
iv) Left to the reader. We use the fact that no reduction is possible on a variable. v) By (iv) and (iii), $\mathrm{Var}_C \cup$ Abort_C satisfies Cand0, Candl and Cand2 if Abort_C satisfies Cand1 and Cand2. Thus, we have to prove Candl, Cand2 for Abort_C. Since C is positive or negative, then $C \neq \perp$, $\mathcal{A}_C\, M$ is not an d redes, and each reduction sequence out of $\mathcal{A}_C M \in \mathrm{Abort}_C$ is a reduction sequence out of M. Then Abort_C satisfies Candl and Cand2 since $M \in SN_\perp$ for each $\mathcal{A}_C M \in \mathrm{Abort}_C$. □

Lemma *3.4 Let A, B be positive or negative types, a an atomic type,* $X \in \mathcal{P}(Term_A)$.
i) X satisfies Cand0 $\Rightarrow$ *Not(X) candidate for* $\neg A$.
ii) $[a]_0$ *is a candidate for a.*
iii) [A], [B] are candidates for A, B $\Rightarrow$ $[A \to B]_0$ *is a candidate for* $A \to B$.

Proof.
i) Since X satisfies Cand0, then $\mathrm{Lambda}^\neg(X)$ satisfies Candl, Cand2 by Lemma 3.3.(ii). By applying 3.3.(v), (iii) in this ortler, we deduce that $\mathrm{Var}_{\neg A} \cup \mathrm{Abort}_{\neg A} \cup \mathrm{Lambda}^\neg(X)$ satisfies Cand0.

Candl, Cand2. Thus, by Lemma 3.2, Not(X) = $\mathbf{Clos}(\mathbf{Var}_{\neg A} \cup \mathbf{Abort}_A \cup \text{Lambda'(X)})$ is a candidate.

ii) By Lemma 3.3.(v) and Lemma 3.2, $[a]_0 = \mathbf{Clos}(\mathbf{Var}_a \cup \text{Abort}_{,,})$ is a candidate.

iii) If *[A]* and *[B]* are candidates for *A* and *B*, then Lambda([A], *[B])* satisfies Candl, Cand2 by Lemma 3.3.(i). By applying 3.3.(v), (iii)in this order, we deduce that $\mathbf{Var}_{A\to B} \cup \mathbf{Abort}_A \cup$ Lambda([A], *[B])* satisfies **Cand0,** Candl, Cand2. We conclude by Lemma 3.2 that $[A \to B]_0 = \mathbf{Clos}(\mathbf{Var}_{A\to B} \cup \text{Lambda}([A], [B]))$ is a candidate. □

Lemma *3.5 Let P be a positive type, X* $\in \mathcal{P}(Term_P)$.

i) X satisfies **Cand0** $\Rightarrow$ *Cont(Not(Not(X))) satisfies Candl, Cand2.*

ii) $[P]_\alpha$ *is a candidate* $\Rightarrow [P]_{\alpha+1}$ *ia a candidate.*

iii) Let β *be a limit ordinal,* $[P]_\alpha$ *a candidate for all* $\alpha < \beta \Rightarrow [P]_\beta$ *in a candidate.*

Proof.

i) Assume that X satisfies **Cand0.** Then, by applying Lemma 3.4.(i) twice, Not(Not(X)) is a candidate. In particular, if $M \in$ Not(Not(X)) and $M \to$ N, then $M \in SN_{\neg\neg P}$ by Candl and N $\in$ Not(Not(X)) by Cand2. We check now that Cont(Not(Not(X))) satisfies Candl, Cand2.

Candl) Each reduction on *CM* is indeed a reduction on *M*, because *M* cannot have type $\neg\neg\bot$ and thus *CM* is not an d) redex. It follows that if CM $\in$ Cont(Not(Not(X))) then $CM \in SN_P$.

Cand2) For the same motivation, if $CM \to N'$, then N' = CM' and $M \to M'$, and therefore $M' \in$ Not(Not(X)), N' = $CM' \in$ Cont(Not(Not(X))).

ii) Assume $[P]_\alpha$ is a candidate.

Then, by point i) above, $\text{Cont}([\neg\neg P]_\alpha) = \text{Cont}(\text{Not}(\text{Not}([P]_\alpha))$ satisfies Candl, Cand2. By Lemma 3.3.(iii) we deduce that $[P]_\alpha \cup \text{Cont}([P]_\alpha)$ satisfies Cand0, Candl, Cand2. We conclude that $[P]_{\alpha+1} = \text{Clos}([P]_\alpha \cup \text{Cont}([P]_\alpha))$ is a candidate by Lemma. 3.2.

iii) By definition, $[P]_\beta = \bigcup_{\alpha<\beta}[P]_\alpha$. Therefore we have to prove that the union of a non- empty increasing chain of candidates satisfies Cand0, Candl, Cand2, Cand3.

Cand0) The condition $[P]_\alpha \supseteq$ Varp are clearly preserved under non-empty unions.

Candl) Simolarly for $SN_P \supseteq [P]_\alpha$.

Cand2) assume $M \in \bigcup_{\alpha<\beta}[P]_\alpha$ and $M \to$ N. Then $M \in [P]_\alpha$ for some α, and $N \in [P]_\alpha$ by Candl for $[P]_\alpha$. Thus, $N \in \bigcup_{\alpha<\beta}[P]_\alpha$.

Cancl3) Assume

$$(\forall Q \in \text{Term}_A)(MN \to_1 Q) \Rightarrow Q \in \bigcup_{\alpha<\beta} [P]_\alpha$$

MN has a finite number of one-step reducts, say $Q_1, \ldots, Q_n$. For each of them we have $Q_i \in \bigcup_{\alpha<\beta}[P]_\alpha$. Therefore, $Q_i \in [P]_{\alpha_i}$ for some $\alpha_i < \beta$. Let $\alpha' = \max\{\alpha_1, \ldots, \alpha_n\} < \beta$ (with $\alpha' = 0$ if n = 0). Since $[P]_\alpha$ is an increasing chain, then $[P]_{\alpha'} \supseteq [P]_{\alpha_i}$ and thus $Q_1, \ldots, Q_n \in [P]_{\alpha'}$. We apply Cand3 to $[P]_{\alpha'}$ and we deduce $MN \in [P]_{\alpha'}$. It follows MN $\in \bigcup_{\alpha<\beta}[P]_\alpha$. □

Lemma 3.6 *Let T be a type and* α *an ordinal. Then* $[T]_\alpha$, $[T]$ *are candidates.*

Proof.
By induction on the definition of $[T]_\alpha$, [T]; i.e., by principal induction on the number of arrows in T, and by secondary induction on α. All the properties required in the inductive steps are in Lemma 3.1, Lemma 3.4 (i), (ii), (iii) and Lemma 3.5 (ii), (iii). □

We are ready to prove now, in the next section, that every term is computable.

3.2 Computability for terms of $\lambda_{C\tau}$-

In order to prove that every term is computable, we have to check that all constructors of the language build computable terms from computable terms. For some connectives, this fact follows by the definition we have given. For variables it follows from the fact that *[A]* is a candidate and from **Cand0.**

Lemma 3.7 *Let A, B be positive or negative types, P a positive type,* $\lambda x.M \in Term_{\neg A}$ *and* $\lambda x.N \in Term_{A\to B}$. *Then :*

i) $(\forall Q \in [A]_\alpha)(M[Q/x] \in SN_\perp) \Rightarrow \lambda x.M \in [\neg A]_\alpha$

ii) $(\forall R \in [A]_\alpha)(N[R/x] \in [B]_\alpha) \Rightarrow \lambda x.N \in [A \to B]_\alpha$

iii) $M \in [\neg\neg P]_\alpha \Rightarrow CM \in [P]_\alpha$

iv) $M \in SN_\perp \Rightarrow \mathcal{A}_A M \in [A]_\alpha$

Proof.
i) $[\neg A] \supseteq \text{Lambda}^\neg([A])$, and $\lambda x.M \in \text{Lambda}^\neg([A])$ by definition of Lambda'.
ii) $[A \to B] \supseteq$ Lambda([A], [B]), and $\lambda x.N \in$ Lambda([A], [B]) by definition of Lambda..
iii) $[P] = \text{Clos}([P] \cup \text{Cont}([\neg\neg P]))) \supseteq \text{Cont}([\neg\neg P])$ by Tarski Theorem, and $CM \in \text{Cont}([\neg\neg P]_\alpha)$ by definition of Cont.
iv) From the definition of $[A]_\alpha$ it is easy to check that $[A]_\alpha \supseteq \text{Abort}_A$ and hence $\mathcal{A}_A M \in [A]_\alpha$ when $M \in SN_\perp$. cl

To check instead that $M \in [A \to B]$ and $N \in [A]$ imply $MN \in [B]$ is more difficult. It will justify the need for the heavy candidate machinery we introduced.

The difficulty to prove $MN \in [B]$ lies in the fact that MN has a functional constructive meaning (it may be reduced by β) but also non-functional ones (it may be reduced by $\mathcal{C}_L$, $\mathcal{C}_R$, $\mathcal{C}'_L$, $\mathcal{C}'_R$, $\mathcal{C}''_L$). Suppose, for instance? that M = CM', and reduce MN by $\mathcal{C}_L$ to $\mathcal{C}\lambda k.M'(\lambda f.k(fN))$. If we try to prove $\mathcal{C}\lambda k.M'(\lambda f.k(fN)) \in [B]$,after a while, because of the presence of $\lambda f \ldots (fN) \ldots$ in $\mathcal{C}\lambda k.M'(\lambda f.k(fN))$, we are reduced to prove $M''N \in [B]$ for any $M'' \in [A \to B]$.
The situation seems to be hopeless; in an attempt to prove $MN \in [B]$, we are reduced to prove $M''N \in [B]$ for all M" $\in [A \to B]$. In other words, the reduction rules we have seem to give a cyclic definition of the constructive meaning of MN. The idea is to break this cycle, by saying that we introduced $M(= CM')$ in $[A \to B]$ after we introduced M' in $[\neg\neg(A \to B)]$, and M'' in $[A \to B]$. This informal idea has been formalized in the definition $[P]_{\alpha+1} =_{Def} \text{Clos}([P]_\alpha \cup \text{Cont}([P]_\alpha))$. This definition says that if we introduced M' in $[\neg\neg P]$ at the stage α, then we introduced CM' in $[P]$ at the stage $\alpha + 1$.

The nest lemma (Lemma, 3.8) characterizes the terms of the form CM occurring in $[P]_\alpha$. Then we will check (Lemmas 3.9 and 3.10) that $M \in [\neg A]$, $N \in [A]$ imply $MN \in SN_\perp$, and finally (Lemma 3.1'2) that $M \in [A \to B]$, $N \in$ [A] imply $MN \in$ [B]. The lasttwo properties axe really hard to prove, but they are required in order to prove that every term is compu ta ble.

Lemma *3.0 Let P be a positive type and α an ordinal. Then:*

$$CM \in [P]_\alpha \Leftrightarrow M \in [\neg\neg P]_{\alpha'}$$

for some $\alpha' < \alpha$.

Proof. $\Leftarrow$) $M \in [\neg\neg P]_{\alpha'}$ for some $\alpha' < \alpha \Rightarrow (CM) \in [P]_{\alpha'+1}$ and $\alpha' + 1 \leq \alpha \Rightarrow (CM) \in [P]_\alpha$ (because the chain $[P]_\alpha$ is increasing).
$\Rightarrow$) We proceed by induction on α. The case $\alpha = 0$ is trivial because $CM \notin [P]_0$ by definition of $[P]_0$.
In the case $\alpha = \alpha' + 1$, if CM $\in [P]_{\alpha'+1} =_{Def}$ Clos($[P]_{\alpha'} \cup$ Cont($[\neg\neg P]_{\alpha'}$)) then, since CM is not an application, either $CM \in [P]_{\alpha'}$ or $CM \in$ Cont($[\neg\neg P]_{\alpha'}$). In the first case we apply induction hypothesis on α' obtaining $M \in [\neg\neg P]_{\alpha''}$ for some $\alpha'' < \alpha' < \alpha' + 1$; in the second one, $M \in [\neg\neg P]_{\alpha'}$ by definition of Cont, and $\alpha' < \alpha' + 1$, as we wished to show.
In the case a is a limit ordinal, if $CM \in \bigcup_{\alpha'<\alpha}[P]_{\alpha'}$ then $CM \in [P]_{\alpha'}$ for some $\alpha' < \alpha$. We apply the induction hypothesis on α' and we obtain $M \in [\neg\neg P]_{\alpha''}$ for some $\alpha'' < \alpha' < \alpha$. □

We check now (in Lemmas 3.9 and 3.10) that $M \in [\neg A]$, $N \in [A]$ imply $MN \in SN_\perp$.

Lemma *3.9 Let P be a positive type and α an ordinal. Then:*
(i) $M \in [\neg\neg P]_\alpha$, $N \in [\neg P]_\alpha \Rightarrow MN \in SN_\perp$
(ii) $M \in [\neg\neg P]$, $N \in [\neg P] \Rightarrow MN \in SN_\perp$

Proof.
(i) By lemma 3.6, $[\neg\neg P]_\alpha$ and $[\neg P]_\alpha$ are candidates. By Candl, M and N have bounds m and n. We prove now that MN $\in SN_\perp$ by induction on m + n. Since $SN_\perp$ is a candidate by Lemma 3. 1, by Cand3 it is enough to prove: $(\forall Q \in \text{Term}_\perp)(MN \to_1 \text{Q}) \Rightarrow \text{Q} \in SN_\perp$.
Assume MN $\to_1$ Q in order to prove Q $\in SN_\perp$. Then there are four cases : either $Q = M_1N$ and M $\to_1 M_1$, or $Q = MN_1$ and N $\to_1 N_1$, or Q = $M'[N/x]$ and $M = \lambda x.M'$ or Q = R and MN = $E[AR]$ (by definition, $\mathcal{C}''_R$ cannot be applied). In the first case, M_1 has a bound $m_1 < m$ and $M_1 \in [\neg\neg P]_\alpha$ by Cand2. In the second one N_1 has a bound $n_1 < n$ and $N_1 \in [\neg P]_\alpha$ by Cand2. In both cases we apply the induction hypothesis and deduce M_1N (or MN_1)$\in SN_\perp$, as required to prove. In the third case, $M = \lambda x.M' \in [\neg\neg P]_\alpha \in$ Not($[\neg P]_\alpha$). Since M is neither an application nor a variable nor an abort, then $M \in$ Lambda$^\neg$($[\neg P]_\alpha$), and we conclude $M'[N/x] \in SN_\perp$ by definition of Lambda,. In the fourth case R is necessarily a subterm of M or N. In both cases $R \in SN_\perp$ since M and N have bounds.
(ii) Straightforward by (i), putting $\alpha = \omega_1$. □

Lemma 3.10 *Let P be a positive type and α an ordinal. Then:*
(i) $M \in [\neg P]$, $N \in [P]_\alpha \Rightarrow MN \in SN_\perp$
(ii) $M \in [\neg P]$, $N \in [P] \Rightarrow$ MN $\in SN_\perp$

Proof.
(i) By lemma 3.6, $[\neg P]$ and $[P]_\alpha$ axe candidates. By Cand1, m and N have bounds m and n. We prove now MN $\in SN_\perp$ by principal induction on α and secondary induction on m + II. Since $SN_\perp$ is a. candidate by Lemma 3.1, by Cand3 it is enough to prove: $(\forall Q \in \text{Term}_\perp)\,(MN \to_1 \text{Q}) \Rightarrow Q \in SN_\perp$.
Assume $MN \to_1 Q$ in order to prove $Q \in SN_\perp$. There are five subcases:

1. Q = M_1N and $M \to_1 M_1$
2. Q = MN_1 and N $\to_1 N_1$N
3. Q = $M'[N/x]$ and $M = \lambda x.M'$ (we applied β)
4. Q = $N'(\lambda a.(Ma))$ and N = CN' (we applied C''_R)
5. Q = R and $MN = E[AR]$ (we applied A).

We check Q $\in SN_\perp$ separately in each case.

1. If $Q = M_1N$ and $M \to_1 M_1$ then M_1 has a bound $m_1 <$ m and $M_1 \in [\neg P]$ by Cand2. By the secondary induction hypothesis we deduce $M_1\ N \in SN_\perp$.
2. If $Q = MN_1$ and N $\to_1 N_1$ then N_1 has a bound $n_1 <$ n and $N_1 \in [P]_\alpha$ by Cand2. By the secondary induction hypothesis we deduce $MN_1 \in SN_\perp$.
3. Assume Q = $M'[N/x]$ and $M = \lambda x.M'$. Then $M \in$ Lambda$^\neg([P])$, because $M \in [\neg P]$ and M is neither an application nor a variable nor an abort. We conclude $M'[N/x] \in SN_\perp$ by definition of Lambda', $N_1 \in [P]_\alpha$ and $[P] \supseteq [P]_\alpha$ (the inclusion holds because P is positive).
4. Assume Q = $N'(\lambda a.(Ma))$ and N = CN'. Then N' $\in [\neg\neg P]_{\alpha'}$ for some $\alpha' \prec \alpha$, by Lemma 3.S and N $\in [P]_\alpha$. To prove $N'(\lambda a.(Ma)) \in SN_\perp$, by Lemma 3.9 it is enough to prove $\lambda a.(Ma) \in [\neg P]_{\alpha'}$. By Lemma 3.%(i), $\lambda a.(Ma) \in [\neg P]_{\alpha'}$ may be proved if we prove $(\forall N'' \in [P]_{\alpha'})\ (MN'' \in SN_\perp)$. This last statement follows by principal inductive hypothesis on $\alpha' < \alpha$.
5. R is necessarily a subterm of M or N. In both cases $R \in SN_\perp$ since M and N have bounds.

(ii) Straightforwardly by (i), putting $\alpha = \omega_1$. □

Lemma 3.11 *For uny positive or negative type A,*

$$M \in [\neg A],\ N \in [A] \Rightarrow MN \in SN_\perp.$$

Proof.
By Lemma 3.9 (if A is negative) or 3.10 (if A is positive).□

We prove now the last and harder lemma of this paper.

Lemma 3.12 *Let A, B be positive or negutive types and α, β be ordinals. If A is a negative type. assume also $\beta = \omega_1$. 'Then:*
(i) $M \in [A \to B]_\alpha, N \in [A]_\beta \Rightarrow MN \in [B]$
(ii) $M \in$ [A → B] , $N \in [A] \Rightarrow MN \in [B]$

Proof.
(i) By lemma. 3.7, $[A \to B]_\alpha$, $[A]_\beta$ and $[B]$ are candidates. By Cand1, M and N have bounds m and n. We prove now $MN \in$ [B] by threefold induction on the indexes α, β, $m + n$. (Actually, the order between the first two indeses does not. matter). By Cand3 it is enough to prove:

$$(\forall Q \in \text{Term}_\perp)(MN \to_1 \text{Q}) \Rightarrow Q \in [B]$$

Assume $MN \to_1$ Q in order to prove Q $\in [B]$. There are five subcases:

1. Q = M_1N and $M \to_1 M_1$

2. Q = MN_1 and N $\to_1$ N

3. Q = $M'[N]$ and $M = \lambda x.M'$ (we applied rule (β))

4. $Q = C\lambda k.M'(\lambda f.k(fN))$, $M = CM'$ and B is positive (we applied rule C_L)

5. $Q = \lambda p.M'(\lambda f.(fN)p)$, $M = CM'$ and B is negative (we applied rule C'_L)

6. $Q = C\lambda k.N'(\lambda a.k(Ma))$, $N = CN'$ and B is positive (we applied rule C_R)

7. $Q = \lambda p.N'(\lambda a.(Ma)p)$, $N = CN'$ and B is negative (we applied rule C'_R).

We check Q $\in$ *[B]* separately for each case.

1. If $Q = M_1N$ and $M \to_1 M_1$ then M_1 has a bound $m_1 <$ m and $M_1 \in [A \to B]_\alpha$ by Cand2. By induction hypothesis on $(m_1 + n)$ we deduce $M_1N \in [B]$.

2. If $Q = MN_1$ and $N \to_1 N_1$ then N_1 has a bound $n1_1 <$ n and $N_1 \in [A]_\beta$ by Cand2. By the induction hypothesis on $(m + n_1)$ we deduce $MN_1 \in [B]$.

3. Assume Q = $M'[N]$ and $M = \lambda x.M'$. Then $M \in$ Lambda([A], *[B]),* because $M \in [A \to B]_\alpha$ and M is neither an application nor a variable nor an abort nor a continuation. We conclude $M'[N/x] \in [B]$ by definition of Lambda and N $\in [A]_\beta$, $[A] \supseteq [A]_\beta$. Remark that $[A] \supseteq [A]_\beta$ holds because either A is a positive type (and the chain $[A]_\alpha$ is increasing) or $\beta = \omega_1$ (and $[A] = [A]_\beta$).

4. Assume Q = $C\lambda k.M'(\lambda f.k(fN))$, M = CM' and B positive. Then $M' \in [\neg\neg(A \to B)]_{\alpha'}$ for some $\alpha' < \alpha$, by Lemma 3.8 and $M \in$ [A $\to B]_\alpha$. We proceed now backwards from our thesis. To prove $C\lambda k.M'(\lambda f.k(fN)) \in [B]$ by Lemma 3.7.(iii) it is enough to prove $\lambda k.M'(\lambda f.k(fN)) \in [\neg\neg B]$. By Lemma 3.7.(ii), $\lambda k.M'(\lambda f.k(fN)) \in [\neg\neg B]$ may be proved if we prove $(\forall Q \in [\neg B])(M'(\lambda f.Q(fN)) \in SN_\perp)$. By Lemma 3.11 and $M' \in [\neg\neg(A \to B)]_{\alpha'}$, $M'(\lambda f.Q(fN)) \in SN_\perp$ may in turn be proved if we prove $\lambda f.Q(fN) \in [\neg(A \to B)]_{\alpha'}$. This last statement, by Lemma. 3.7.(i), is implied by $(\forall M'' \in [A \to B]_{\alpha'})(Q(M''N) \in SN_\perp)$. Since Q $\in [\neg B]$, by Lemma 3.11 all we have to prove is $(\forall M'' \in [A \to B]_{\alpha'})(M''N \in [B])$. This last statement may be obtained by the principal induction hypothesis on $\alpha' < \alpha$.

5. Assume Q = $\lambda p.M'(\lambda f.(fN)p)$ and M = CM', with $B = \neg P$ for some positive type P. Then $M' \in [\neg\neg(A \to B)]_{\alpha'}$ for some $\alpha' < \alpha$, by Lemma 3.S and $M \in [A \to B]$,. *We* proceed now backwards from our thesis. To prove $\lambda p.M'(\lambda f.(fN)p) \in [B] = [\neg P]$ by Lemma 3.7.(i) it is enough to prove (VQ $\in [P])(M'(\lambda f.(fN)Q) \in SN_\perp)$. By Lemma 3.11 and $M' \in [\neg\neg(A \to B)]_{\alpha'}$, $M'(\lambda f.(fN))Q \in SN_\perp$ may in turn be proved if we prove $\lambda f.(fN)Q \in [\neg(A \to B)]_{\alpha'}$. This last statement, by Lemma. 3.7.(i), is implied by $(\forall M'' \in$ [A $\to B]_{\alpha'})(\ (M''N)Q \in SN_\perp)$. Since Q $\in [P]$, by Lemma. 3.11 all we have to prove is $(\forall M'' \in [A \to B]_{\alpha'})(M''N \in [B] = [\neg P])$. This last statement may be obtained by the principal induction hypothesis on $\alpha' < \alpha$.

6. Assume Q = $C\lambda k.N'(\lambda a.k(\ Ma))$, $N = CN'$ and B positive. Then $N' \in [\neg\neg A]_{\beta'}$ for some $\beta' < \beta$, by Lemma. 3.8 and $N \in [A]_\beta$. Remark that, for the restriction we put on typing, A must be a. positive type, and therefore rue *did not assume* $\beta = \omega_1$. We proceed now backwards from our thesis. To prove $C\lambda k.N'(\lambda a.k(Ma)) \in [B]$ by Lemma 3.7.(iii) it is enough to prove $\lambda k.N'(\lambda a.k(Ma)) \in [\neg\neg B]$. By T,emma 3.7.(ii), $\lambda k.N'(\lambda a.k(Ma)) \in [\neg\neg B]$ may be

proved if we prove $(\forall Q \in [\neg B])(N'(\lambda a.Q(Ma)) \in SN_\perp)$. By Lemma 3.9 and N' $\in [\neg\neg A]_{\beta'}$, $N'(\lambda a.Q(Ma)) \in SN_\perp$ may in turn be proved if we prove $\lambda a.Q(Ma) \in [\neg A]_{\beta'}$. This last statement, by Lemma 3.7.(i), may be deduced from (VN" $\in [A]_{\beta'}$)$(Q(MN'') \in SN_\perp)$. Since Q $\in [\neg B]$, by Lemma 3.11 'all we have to prove is (VN" $\in [A]_{\beta''}$) (MN" $\in [B]$). This last statement may be obtained by the secondary induction hypothesis on $\beta' < \beta$ (in order to apply inductive hypothesis to $\beta' < \beta$, it is crucial that we did not assume $\beta = \omega_1$).

7. Assume Q = $\lambda p.N'(\lambda a.(Ma)p)$ and N = CN' with $B = \neg P$. Then $N' \in [\neg\neg A]_{\beta'}$ for some $\beta' < \beta$, by Lemma 3.8 and N $\in [A]_\beta$. Remark that, for the restriction we put on typing, A must be a positive type, and therefore we did *not assume* $\beta = \omega_1$. We proceed now backwards from our thesis. To prove $\lambda p.N'(\lambda a.(Ma)p) \in$ [B] = $[\neg P]$ by Lemma 3.7.(i) it is enough to prove (VQ $\in [P]$)$(N'(\lambda a.(Ma)Q) \in SN_\perp)$. By Lemma 3.9 and $N' \in [\neg\neg A]_\beta$, $N'(\lambda a.(Ma)Q) \in SN_\perp$ may in turn be proved if we prove $\lambda a.(Ma)Q \in [\neg A]_{\beta'}$. This last statement, by Lemma 3.7.(i), may be deduced from (VN" $\in [A]_{\beta'}$)$((MN'')Q \in SN_\perp)$. Since Q $\in [P]$, by Lemma 3.11 all we have to prove is (VN" $\in [A]_{\beta'}$)$(MN'' \in$ [B] = $[\neg P])$. This last statement may be obtained by the secondary induction hypothesis on $\beta' < \beta$ (in order to apply inductive hypothesis to $\beta' < \beta$, it is crucial that we did not assume $\beta = \omega_1$).

(ii) Straightforwardly by (i), putting $\alpha = \beta = \omega_1$. □

3.3 **The** result

We now ready to prove a' Soundness Theorem and to deduce Strong Normalization from it. We only need a last definition before.

Definition 3.5 *Let M be any term.*

(i) A substitution is uny mup from a finite set of variables to the terms.

(ii) A substitution σ is on M if the free variables of M ure all in the domain of σ.

(iii) If a is a substitution on M, we denote by a(M) the result of replacing each x free in M by $\sigma(x)$.

***(iv) A** substitution σ is computable if $\sigma(x)$ is computable for all variables x in the domain of σ.*

Theorem 3.1 (Soundness) *Let M be any term and σ a substitution on it. Then:*
σ is computable $\Rightarrow \sigma(M)$ is computable

Proof.
By induction on AI.

- M is a variable.
  The thesis holds by definition of computable substitution.
- $M \equiv \lambda x M_1$
  We apply Lemma 3.7.(i) or 3.7.(ii), if the type of M has the form $\neg A$ *or* $A \to B$ respectively.
- $M \equiv M_1 M_2$
  We apply Lemma 3.11 or 3.12, if the type of M_1 has the form $\neg A$ or $A \to B$ respectively.

- $M \equiv CM_1$
  We apply Lemma 3.7.(iii).
- $M \equiv AM_1$
  We apply Lemma 3.7.(iv). □

Corollary 3.1 (Strong Normalization) *Every term M of $\lambda_{C\tau}$-strongly normalizes.*

Proof. Consider the identical substitution on M, defined by $id(x) = x$ for each x free in M. The substitution id is computable because $x^A \in [A]$ by Cand0. Therefore by the Soundness Theorem $M(= id(M))$ is computable. Thus, $M \in [A]$ for the type A of M. Then, by Lemma 3.6 and Cand1, M strongly normalizes. □

Acknowledgements

We are grateful to Mariangiola Dezani for her constant support and gentle guidance, and to Mario Coppo for helpful discussions and his careful reading of an earlier draft.

The first author wishes to express his gratitude also to Paola Fochesato and Pierpaolo Fiorletta for their steadfast encouragement.

References

1. Barbanera F., Berardi S. Witness Extraction in Classical Logic through Normalization. To appear in *Proceedings of BRA-LF workshop,* Cambridge University Press.
2. Barbanera F., Berardi S. A constructive valuation interpretation for classical logic and its use in witness extraction. To appear in *Proceedings of Colloquium on Trees in Algebra and Programming (CAAP), 1992.*
3. Clinger W., Rees J. The *revised*^{3} report on the algorithmic language scheme. *SIGPLAN Notices,* 21(12):37-79, 1986.
4. M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control. Technical report 100, University of Rice, Houston, 1989. To appear in *Theoretical Computer Science.*
5. M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba, Reasoning with continuations. In "Proceedings of the First Annual Symposium on Logic in Computer Science", pages 131-141,1986.
6. Timothy G. Griffin. A formulas-as-types notion of control. In"Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 1990.
7. Murthy C. Extracting constructive content from classical proofs. Ph.d. thesis, Department of Computer science, Cornell University, 1990.
8. C. Murthy. An evaluation semantic for classical proof. In "Proceedings of the Fifth Annual Symposium on Logic in Computer Science" 1991.
9. Murthy C. Finding the Answers in Classical Proofs: A Unifying Framework. To appear in *Proceedings of BRA-LF workshop,* Cambridge University Press.

Three Monads for Continuations *

Richard B. Kieburtz Borislav Agapiev James Hook
Oregon Graduate Institute of Science & Technology
19600 N.W. von Neumann Dr.
Beaverton, OR 97006

e-mail : *lastname*@cse.ogi.edu

1 Monads capture semantic structure

We propose three monads that express the structure of different modes of continuation semantics. The first is the familiar CPS semantics, the second is a semantics for languages with first-class continuations, and in the third we have 'composable continuations' that are useful to express the semantics of backtracking such as occurs in the computations of logic programs. The third structure is not actually a monad, as the left identity law fails for reasons that we discuss. The object map of each monad, if expressed as a formula of propositional logic, forms the hypothesis of an implication from which one can derive full classical propositional logic.

In each monad (or pre-monad), certain morphisms of an underlying category can be 'taken for granted', rather than constructed. These morphisms are respectively, eval, the evaluator of applicative expressions, call/cc, a meta-language analog of the call/cc control primitive of Scheme, and a new (but related) morphism that we call eval/cc.

1.1 Monads of a Cartesian-closed category

As a model for programming language semantics, we assume an underlying Cartesian-closed category. The intended interpretation is that objects of the category correspond to types and morphisms to functions. State is easily accommodated in such a model [11, 10]. We shall use the following characterization of a monad [8]

Definition 1: A Kleisli triple $(T, \eta, (\_)^*)$ in a category C consists of

- an object mapping function $T : Obj(C) \rightarrow Obj(C)$,
- a natural transformation called the unit, $\eta_X : X \rightarrow TX$,

*The research reported here was supported in part by the National Science Foundation under grant No. CCR-9101721.

- a natural extension operation that takes each morphism $f : X \to TY$ to a morphism $f^* : TX \to TY$ in C.

These components of a monad must satisfy three laws:

(K1) $\eta_X^* = id_{TX}$

(K2) $f^* \circ \eta = f$

(K3) $(g^* \circ f)^* = g^* \circ f^*$

□

Laws (K1) and (K2) express that the unit is respectively, a left and a right identity with respect to Kleisli composition. Law (K3) expresses that natural extension is associative with respect to morphism composition. We shall call a monad-like structure a left or a right pre-monad if it satisfies (K3) and one of the identity laws, (K1) or (K2), but the other identity is not assured.

A morphism $h : X \to Y$ of the underlying category can be 'lifted' to a T-monadic morphism by composition on the left with the unit of the monad, $\eta_Y \circ h : X \to TY$. Such morphisms are called the proper, (or existing) morphisms of T. The natural extension of proper morphisms provides a mapping of morphisms $X \to Y$ to $TX \to TY$ which together with the object mapping function constitutes a functor $T : C \to C$. The more interesting morphisms of a monad are those of types $X \to TY$ that are non-proper. For each of the monads we consider, we shall be interested in the interpretation given to its non-proper morphisms.

1.2 What is a monad for continuations?

Each of the monadic structures studied here can be used to transform a direct semantics for the X-calculus to a call-by-value semantics that uses continuations explicitly. As the language is extended, we do not expect that every function will denote a proper morphism in the category. Hence, functions given the type $X \to Y$ in the language will correspond to morphisms from X to TY in the category, i.e. they will map values to computations. Furthermore, an expression representing a function value acquires a type $T(X \to TY)$. These observations yield what has been called the ***Kleisli*** *interpreter* [2]:

$$\begin{aligned} [\![x]\!]\rho &= \eta(\rho\, x) \\ [\![\lambda x.M]\!]\rho &= \eta(\lambda v.[\![M]\!]\rho[x \mapsto v]) \\ [\![M\ N]\!]\rho &= (\lambda f.f^*([\![N]\!]\rho))^*([\![M]\!]\rho) \end{aligned}$$

The natural extension of a function in the monad allows substitution of computations, rather than values, for argument variables. This is the essence of continuation semantics.

1.3 The continuation-passing monad

The functions of CPS semantics are captured in the monad whose object function, unit and natural extension operation are:

$$\begin{aligned} T\,X &= (X \to A) \to A \\ \eta_X &= \lambda x \lambda c.cx \\ f &= \lambda t.\lambda c.t(\lambda x.fxc) \\ &\text{where } f : X \to TY \end{aligned}$$

We call this the CPS monad. It has previously been called the monad of continuations [10] but, as we shall see, it is not the only interesting monadic structure that captures computation with continuations.

If C is to be a nontrivial Cartesian-closed category, then the naturality required of the unit and the extension restrict the choice of object A. Intuitively, A should correspond to a universal type for final answers; a type on which we can assume no interesting operations to be defined. Formally, A must be an involution object [9, 7] of C, satisfying two conditions [1]

(I1) the function space $[A \to A]$ is a final object in C;

(I2) $\forall A \in Obj(\mathbf{C})$ there is a monic arrow $\kappa_X : X \to A$.

Continuations are modeled as arrows $X \to A$. Condition (12) guarantees that there is a continuation that can distinguish the elements of X from one another.

In the propositions-as-types analogy between intuitionistic propositional logic and the simply typed λ-calculus, X-terms of type t correspond to proofs of the formula corresponding to t in the logic. Closed lambda terms correspond to proofs of tautologies. Griffin [5] observed that the analogy extends to one relating classical logic to a X-calculus extended with typed continuations, and used the analogy to suggest types for control operators.

An involution object is analogous to the absurdity proposition of an intuitionistic logic [5]. An object TX is analogous to a double-negation proposition, $\neg\neg X$, in intuitionistic logic. The formula $\neg\neg X \Rightarrow X$, when added as an axiom scheme, yields classical logic. Analogous to this formula is a morphism $\mathbf{eval}_X : TX \to X$ in the category C. For proper computations of T it satisfies:

$$\mathbf{eval}(\lambda c.cx) = x$$

Such a morphism cannot be defined as a closed &expression, i.e. it does not necessarily exist as a consequence of the Cartesian-closed property of C. Although $\mathbf{eval}_X$ is natural in X it is not universal unless every arrow $Y \to TX$ in C is proper for the monad T. Its formal introduction rule is:

$$\frac{\Gamma \vdash t : TX}{\Gamma \vdash \mathbf{eval}_X\, t : X}$$

[1] The reader should note that involution does not imply isomorphism of objects X and TX. Such an isomorphism would have as a consequence that if C is Cartesian-closed, it is an order category, in which there is at most one arrow between any pair of objects [6].

The CPS monad internalizes as objects TX the morphisms that map X-accepting continuations to final results. Such objects are sets of 'latent computations' that provide semantics for applicative expressions. If $f' : X \to Y$ is a morphism of C, then the proper morphism $f = \eta_Y \circ f' : X \to TY$ satisfies the equation

$$f\,XC = c(f'x)$$

Non-proper morphisms of this monad are those whose codomain element may represent a computation that discards the nominal result continuation and instead uses a different continuation to effect a tail-call or to raise an exception, or which diverges.

1.4 The monad of control alternatives

The second monad we consider is motivated by the desire to provide semantics to expressions abstracted on a continuation variable. The constituents of the monad are:

$$\begin{aligned} \mathsf{S}\,X &= (X \to A) \to X \\ \eta_X &= \lambda x.\lambda c.x \\ f^* &= \lambda s.\lambda c.f(s(\lambda x.c(fxc))c \\ &\text{where } f : X \to SY \end{aligned}$$

As before, A is required to be an involution object, and an object $[X \to A]$ is interpreted as a type of X-accepting continuations.

The intuitionistic formula analogous to an object SX is $\neg X \Rightarrow X$, which in classical logic is abbreviated as $X \vee X$. A morphism $SX \to X$ can be interpreted as evaluating a computation that might produce a value of type X in two different ways, either by a direct evaluation, ignoring the continuation argument, or by invoking the argument continuation. The analogy with a disjunctive formula of logic hints that SX may be related to a disjoint sum, X + X. This is indeed the case, provided there is added to the set of monad morphisms a constructor $\mathcal{A}_X : A \to X$, called 'abort' [4]. Then we can define

$$\begin{aligned} inl &= \eta = \lambda x.\lambda c.x \\ inr &= \lambda x.\lambda c.\mathcal{A}(cx) \end{aligned}$$

The discriminator is

$$\mathbf{case}(s,f,g) = \lambda c.f(s(\lambda x.c(g\,x\,C)))C$$

in which s : SX, f : X → Y and g : X → Y. Notice the similarity in form between the discriminator and the natural extension of a function in the monad S,

$$f^* = \lambda s.\mathbf{case}(s, f, f).$$

It is informative to compare this formulation with Griffin's construction of disjunctive types [5] in the CPS monad. That construction requires the explicit addition of both the operator $\mathcal{A}$ and of Felleisen's control operator [4], C, while in the monad S we need add only $\mathcal{A}$ as an

explicit operator. However, since A is conventionally defined in terms of C, an independent axiom is needed for A if we are to define it without C. The necessary axiom is

$$(\forall x : X)\ (\forall c, c' : X \to A)\ c'(\mathcal{A}_X(c\,x)) = c\,x$$

Abbreviating $Xs.case(s, f, g)$ as $[f, g]$, it is now easy to check that the axioms of a coproduct hold:

$$[f,g] \circ inl = f \qquad [f,g]\ 0\ inr = g \qquad [id,\ inr] = id$$

Thus, types SX indeed become coproducts if the met a-language includes the abort operator $\mathcal{A}_X$.

Additionally, one can simply postulate a constructor that injects expressions of type SX into a X-calculus. The introduction rule is

$$\frac{\Gamma \vdash s : SX}{\Gamma \text{ t-} \mathbf{call/cc}_X \text{ s} : \text{X}}$$

The explanation of call/cc is that when applied to an abstraction expression, $\lambda c.e$, binds the abstraction variable, c, to the current continuation. Any subexpression of the form c e' is interpreted as a 'throw' of the value of expression e' to the bound continuation.

But what if the value of e' is itself constructed with call/cc? The semantics of composite expressions in this monad are explained by the Kleisli composition, i.e. by using the natural extension in the monad S of functions that may either produce normal values (the arrows proper for the monad) or values constructed with call/cc (the non-proper arrows).

In Scheme, call/cc has been lifted from its status as a semantic operator of the **meta**-language to become a syntactic operator of the programming language. The Kleisli interpreter for the monad S can be extended to account for this language construct:

$$[\![\mathbf{call/cc}\ \lambda x.M]\!]\ \rho = \lambda c.([\![M]\!]\ \rho[x \mapsto c])\ c$$

To complete the analogy with formulae of logic, note that the logical formula $(\neg X \Rightarrow X) \Rightarrow X$ is Peirce's law, also sufficient to yield full classical logic when added to intuitionistic logic as an axiom scheme. This formula corresponds to the type of $\mathbf{call/cc}_X : SX \to X$.

1.5 The pre-monad of composable contexts

The third structure is intended to provide a complete foundation for a semantics of logic programs, or of a language with the prompt and control primitives introduced by Felleisen [3] (or the reset and shift primitives of [1]). This structure is a composite of the two previous ones, with constituents:

$$\begin{aligned} RX &= T(SX) = (((X \to A) \to X) \to A) \to A \\ \eta_X &= \eta^T_{SX} \circ \eta^S_X = \lambda x.\lambda h.h(\lambda c.x) \\ f^* &= \lambda r.\lambda h.r(\lambda s.f(s(\lambda x.fxh))h) \\ &\text{where } f : X \to RY \end{aligned}$$

This structure is not a monad, as the left identity law (K1) fails, but it is a right pre-monad. The left identity law would be provable if elements of type SX were restricted to those constructed by application of η^S_X, but then the monad R would be isomorphic to the CPS monad. We conjecture that the left identity law may also be provable in a category without fixpoints, which would imply that it is connected with the uniform termination problem for R-computations.

An object RX is a space of computations that take SX-expecting continuations to final results. We call an SX-accepting continuation an X-expecting context. A context supplies its SX-typed argument with both an X-expecting continuation for a result produced by normal evaluation and a second continuation of the same type for use if the evaluation aborts. Thus an aborted computation need not escape to the 'top level', but may backtrack. Aborting a computation with an alternate continuation is equivalent to continuing the computation in another context. This intuition is summarized in the CPS transformation of SX-typed expressions:

$$[\![inl\ x]\!]_T\ h = h(inl\ x)$$
$$[\![inr\ x]\!]_T\ h = h_0(inl\ x)$$

where h_0 is a context constant, or initial context. (There is no closed X-term of type SX $\rightarrow A$.)

A semantics of either applicative or relational expressions built with this monad allows contexts to be composed incrementally. Incremental composition of continuations was not possible in either of the monads T or S, because continuations do not compose as ordinary functions. It is possible in R, because higher-order continuations are available as contexts. The Kleisli composition in R allows context abstractions to occur as arguments of functions, in effect subsuming higher-order CPS transformations.

A morphism $RX \rightarrow$ X can be interpreted as evaluating a latent computation that uses an SX-accepting continuation to produce an X-typed result. A morphism of this monad is $\mathbf{eval/cc}_X : RX \rightarrow$ X (evaluate-in-current-context), which is defined by the rule:

$$\text{eval/cc}(r) = \text{eval}(\lambda c.r(\lambda s.c(\text{sc})))$$

where **eval** is the morphism that supplies the immediate continuation to a computation of type TX.

The formula of propositional logic analogous to the type of eval/cc, namely $\neg\neg(\neg X \Rightarrow X) \Rightarrow X$, also yields classical logic when added to intuitionistic logic as an axiom.

1.6 A hierarchy of monads

The monad T provides a semantics in which sequential computation is made explicit. It allows the definition of first-class suspensions. If e is an expression, then $\lambda c.ce$ is a suspension, where c is a continuation variable, and e contains no free occurrence of c. To evaluate a suspension, a current continuation is supplied by eval, by forming an expression $\mathbf{eval}(\lambda c.ce)$. In a call-by-value language, suspensions are made explicit when delayed evaluation is specified. In a non-strict programming language, the notation for suspensions and for their evaluation is implicit.

In the monad S, the restriction that e contains no free occurrence of a continuation variable is lifted. A continuation abstraction is evaluated by **call/cc**($\lambda c.e$) which binds the continuation variable to the current continuation. An expression $\lambda c.e$ is not considered to be a suspension, but one that may depend upon an alternate continuation. The normal result continuation is implicitly furnished whenever an expression is evaluated.

The pre-monad R is a CPS monad of control alternatives. It subsumes both suspended computations and backtracking control. In it, expressions may be abstracted on variables of types $X \rightarrow RX$ which represent context transformations. Application of such a variable to an argument expression can represent a backtrack with that expression or the raising of an exception or the extension of the current context with a previously specified context fragment. We conjecture that the monad R will provide a framework suitable for the semantics of logic languages as well as functional languages with explicit control primitives.

As an example, in Section 2 we give a semantics in the monad RX to the Scheme-like language enriched with shift and reset that was used by Danvy and Filinski. We shall see that these primitives can be realized with eval/cc.

2 Semantics of applicative expressions with shift and reset

The shift/reset (S/R) language we consider here is essentially that given in [1], except for a minor variation in the definition of the shift operator. It is a language of lambda expressions augmented with (strict) operator symbols, a conditional expression, and control operations that allow an alternate context for control to be specified, invoked, or composed with a bounded context segment. There are three explicit control primitives: .

- reset sets a contextual control point. A reset is indicated by angle brackets. When an expression is bracketed, ... ⟨E⟩ ..., the context outside the brackets is marked as accessible. Evaluation of the bracketed expression may depend upon this context in interesting ways, if it contains occurrences of either of the other two control primitives. If E does not refer to these primitives, it is evaluated in the surrounding context just as if the reset brackets were not present. A top-level program is implicitly bracketed by a reset that marks an initial context.

- abort is an abstraction operator that binds a variable to the immediate context of the abstraction. When a variable bound to an abort is applied to an argument expression, the argument is evaluated in the bound context, ignoring the immediate context of the application.

- shift is an abstraction operator that binds a variable to the bounded context segment found between the immediate context and the context mark set by the enclosing reset brackets. Application of a variable bound to a shift extends the current context of the application with the bounded context segment. Repeated applications iterate the extension. A shift abstraction is trivial if there is no occurrence of the bound variable in the body. Then the body of the abstraction is evaluated in the current context just as

if the abstraction were not present. Thus elaboration of a shift abstraction duplicates the bounded context segment one or more times. In the version of the S/R language presented in [1], a trivial shift abstraction is equivalent to an abort with the body of the abstraction, thus elaboration of a shift abstraction duplicates the bounded context zero or more times.

A semantics of the S/R language is given below. The meaning function is

$$\mathcal{E}[\![\_]\!] : \mathbf{Expr} \rightarrow Env \rightarrow R(Value)$$

where $\mathbf{Env} = Identifier \rightarrow Value$. Variables appearing in the formulae are typed as:

$$\begin{array}{rl} h, h', h'' & : SX \rightarrow A \text{ or } SY \rightarrow A \\ s, s_1, s_2 & : \text{So, where } \alpha \text{ ranges over types,} \\ c, c' & : X \rightarrow A \text{ or } Y \rightarrow A \\ x, v & : \mathrm{X} \\ \pi & : \mathrm{X} \rightarrow \mathrm{Y}, \text{ a strict operator} \\ f, f' & : X \rightarrow RY \end{array}$$

The semantic equations for applicative expressions with **shift** and **reset** are:

$$(unit) \qquad \mathcal{E}[\![x]\!]\rho\, h = h(\lambda c.\rho[\![x]\!])$$

$$(pi) \qquad \mathcal{E}[\![\pi]\!]\rho\, h = h(\lambda c.\lambda v.\lambda h'.h'(\lambda c'.\pi\, v))$$

$$(if) \qquad \mathcal{E}[\![\mathbf{if}(E_0, E_1, E_2)]\!]\rho\, h = \mathcal{E}[\![E_0]\!]\rho\,(\lambda s.h(\lambda c.s(\lambda b.b = true \rightarrow \mathcal{E}[\![E_1]\!]\rho\, h; \mathcal{E}[\![E_2]\!]\rho\, h)))$$

$$(app) \qquad \mathcal{E}[\![E_1\, E_2]\!]\rho\, h = \mathcal{E}[\![E_1]\!]\rho\,(\lambda s_1.\mathcal{E}[\![E_2]\!]\rho\,(\lambda s_2.\ \mathbf{let}\ f' = s_1\,(\lambda f.\mathcal{E}[\![E_2]\!]\rho(\lambda s.f\,(s(\lambda x.f\,x\,h))\,h))\ \mathbf{in}\ f'\,(s_2\,(\lambda x.f'\,x\,h))\,h))$$

$$(abs) \qquad \mathcal{E}[\![\lambda x.E]\!]\rho\, h = h(\lambda c.\lambda v.\mathcal{E}[\![E]\!]\rho[[\![x]\!] \mapsto v])$$

$$(abort) \qquad \mathcal{E}[\![\epsilon k.E]\!]\rho\, h = \mathcal{E}[\![E]\!]\rho[[\![k]\!] \mapsto \lambda v.\lambda h'.h(\lambda c.v)]\, h$$

$$(shift) \qquad \mathcal{E}[\![\xi k.E]\!]\rho\, h = \mathcal{E}[\![E]\!]\rho[[\![k]\!] \mapsto \lambda v.\lambda h'.h'(\lambda c.\mathbf{eval/cc}(\lambda h''.h(\lambda c'.v)))]\, h$$

$$(reset) \qquad \mathcal{E}[\![\langle E\rangle]\!]\rho\, h = h(\lambda c.\mathbf{eval/cc}(\mathcal{E}[\![E]\!]p))$$

Expressions like $\lambda v.\lambda h'.h'(\lambda c'.\pi\ \mathrm{v})$, which occurs in (pi) and $\lambda v.\mathcal{E}[\![E]\!]\rho[[\![k]\!] \mapsto v]$, which occurs in *(abs),* represent normal values that have functional types, $\mathrm{X} \rightarrow RY$. Formulas *(unit), (abs)* and *(app)* are calculated directly from the definition of the Kleisli interpreter for this monad. Formulas *(pi)* and *(if)* are similarly obtained from an extension of the intepreter. The formula *(abort)* is a straightforward formalization of the informal description of the abort operator.

We shall not attempt to give an operational interpretation of the (shift) and (reset) semantics, but note that each uses the eval/cc operator in a different way. In *(shift),* the argument expression given to eval/cc discards its context argument. Thus **eval/cc** is used here simply to coerce to a value a computation that uses the context of the shift abstraction as an immediate context. The value so produced is then injected by the unit of the monad to continue computation in a context surrounding an application of the bound variable. The use of eval/cc in (reset) distributes the normal continuation of the immediate context of the reset brackets as both the normal and the alternate continuation to be used in evaluating *E.* This is how a surrounding context is made accessible.

The reader may wish to contrast these semantic equations with those given in [1]. There the semantics is obtained by iterating the CPS transformation twice. Thus we might expect meanings to have types $T(TX)$ and indeed they do, up to an isomorphism of Cartesian closed categories. The authors point out that the CPS transform can be further iterated if one wishes to accommodate nested reset brackets in a language. The semantic equations then become rather unwieldy, although orderly. This is because T^2 is not a monad. By giving semantics in the monad R, iteration of the CPS transform is rendered unnecessary.

3 Acknowledgements

We wish to thank the program committee for the interest shown in this work by several pages of technical comments. We are particularly grateful to Olivier Danvy for helpful and stimulating discussions.

References

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming,* pages 151-160, June 1990.

[2] Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 1991.

[3] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages,* pages 180-190, January 1988.

[4] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science,* 52(3):205–237, 1987.

[5] Timothy Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages,* pages 47-58, January 1990.

[6] Hagen Huwig and Axel Poigné. A note on inconsistences caused by fixpoints in a Cartesian closed category. *Theoretical Computer Science,* 73:101–112, 1990.

[7] Richard B. Kieburtz and Borislav Agapiev. What is an abstract machine? Technical Report CSE-91-011, Department of Computer Science and Engineering, Oregon Graduate Institute, 1991.

[8] Saunders **MacLane.** *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[9] Narciso **Marti-Oliet** and Jose Meseguer. Duality in closed and linear categories. Technical Report SRI-CSL-90-01, SRI International, February 1990.

[10] Eugenio Moggi. Notions of computations and monads. *Information and Computation,* **93(1):55–92,** July 1991.

[11] Philip Wadler. Comprehending monads. In ***Proc.*** *1990 ACM Conference on Lisp and Functional* ***Programming,*** pages **61–78, 1990.**

Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work *

Chetan R. Murthy†
Projet FORMEL
INRIA-Rocquencourt
B.P. 105
78153 Le Chesnay Cedex, FRANCE
murthyQmargaux.inria.fr

April 20, 1992

Abstract

Control operators are an important programming language feature, and are being incorporated into more and more functional programming languages. It is becoming clear that such operators permit beautiful, elegant solutions to many difficult programming problems. Unfortunately, it is difficult to statically type-check these operators. In this paper, we focus on a particular hierarchy of control operators [DF90] and point out how this hierarchy can be given an expressive type system via A-tra.nsla.tion. The type system is expressive in that one can give types to programs which intuitively model their operational behaviour, and do not restrict their usage in order to achieve type-safety. We define the hierarchy of control operators and delimiters, as in [DF90], and present a monomorphic simple type system. We proceed with a. set of local reduction rules which capture the global evaluator, in the manner of [FFKD86], and discover the evaluator also. With the A-translated type-system, we prove a subject reduction theorem, showing that reduction, as well as program-contraction, indeed preserves typing. This work highlights the importance of A-translation in providing accurate type systems for complex control-operakor languages.

1 Introduction

Control-operator languages [FFKD86, Fel87, CR86] are becoming more and more popular, both for the users of programming languages, and for designers. Hence, it becomes important, to have efficient., versatile systems for reasoning about programs written in such languages, and, secondly, to have useful, genera&purpose techniques for constructing such reasoning systems. In this paper, we consider a particular control-operakor language - the hierarchy of Danvy & Filinski [DF90]. This hierarchy is defined by a semantic interpreter, written in estended continuation-passing-style.

*An expanded version of this paper, containing full proofs, can be had (at least until the end of 1992) via anonymous FTP at `ftp.cs.cornell.edu:pub/murthy/cw92-big-version.dvi`.

† Supported in part by an NSF graduate fellowship and NSF grant CCR-8616552 and ONR grant N0001 4-88-K-0409 and ESPRIT Basic Research Action "Logical Frameworks"

As a result, it is sometimes difficult to understand what a program written in the hierarchy does. Further complicating matters is the fact that the "intuitive semantics" (for many people) of control-operator languages is based on C-rewriting machine technology [FF86]. Thus, when we write down a C-rewriting machine for the hierarchy, there rests the immediate problem of verifying adequacy between the two semantics.

To ameliorate these problems, we use A-translation techniques [Fri78, Mur91], and devise a type system for the language which accurately reflects the runtime behaviour of terms. The type system is operationally sound, in that evaluating a well-typed program in the semantic interpreter produces a well-typed result. Using this type system as a guide, we then construct a set of local rewrite rules, in the spirit of Felleisen's. Each is verified by the semantic interpreter, but it becomes obvious that the type system is an indispensable guide to finding the rules.

With the rules, and the type system, in hand, we already have a considerable improvement over the original situation. The type system provides intuitive information about the runtime behaviour of terms. We then apply the techniques of Felleisen and Friedman [FF86], and define an abstract machine for the hierarchy [1] in the spirit of the CEK machine [Rey72], and then proceed to concretize it completely, yielding a C-rewriting machine. This machine is then proven equivalent to the original continuation semantics. Moreover, a slight variation of the presentation of the machine yields the intuitive C-rewriting semantics.

Plan

The plan of this paper is as follows. We begin by presenting a quick overview of Danvy & Filinski's hierarchy, esplaining its connections with other hierarchies, and why we chose this particular system of control operators for our study. The following sections discuss pseudo-classical typing in the case of $\lambda + \mathcal{C}$, the extension with a "prompt" operator, pseudo-classically typing the entire hierarchy, discovering local reduction rules, and discovering a C-rewriting machine. Then we conclude. ,

Preliminaries

For want of a better place, we introduce here a few basic definitions. We define a value as a basic constant, pair, injection of values, or a X-abstraction. A concrete value is one devoid of λ-abstractions. A concrete type is one whose values are devoid of λ-abstractions.

2 Danvy & Filinski's Hierarchy of Control Operators

Danvy & Filinski's hierarchy is a natural estension of the call-by-value language $\lambda+\mathcal{C}$, discovered by Felleisen, Friedman, and their co-workers at Indiana [FFKD86]. Thus, we begin with the language $\lambda + \mathcal{C}$:

Definition 1 ($\lambda + \mathcal{C}$) Term Set: To the CBV λ-calculus, add two new unary term-forming operators. $\mathcal{A}$ neither binds nor creates free variables; $\mathcal{C}$ binds a. free variable, and creates no new ones. Terms V (resp. M, N), perhaps subscripted, are values (resp. arbitrary expressions).

Evaluation (denoted $\triangleright_{\mathcal{C}v(LR)}$):

[1] To give credit where credit is due, we constructed this abstract machine only after private com munication with Danvy, who assured that such was possible.

Program Contraction Rules:

$$\begin{array}{rcl} E[(\lambda x.b)V] & \rhd_1 & E[b[V/x]] \\ E[\mathcal{C}k.M] & \rhd_1 & M[(\lambda x.\mathcal{A}E[x])/k] \\ E[\mathcal{A}M] & \rhd_1 & M \end{array}$$

Evaluation Contexts (Call-by-Value, Left-to-Right):

$$E = [\,] \mid E(N) \mid V(E)$$

Thus, the action of a C-term is to grab the evaluation context, and package it up into a procedure-like object, which is substituted for the bound variable in its subterm argument. When this procedure-like object is applied to some value, the then-current evaluation context is erased, in favor of the saved context. One puzzling thing about this operator, though, is that the transitions for C and d are defined with respect to the entire program being evaluated; hence, it is difficult to produce observational congruences which capture these transitions. The essential difficulty is that the "limit" of the effect of the control operator-is the "top" of the program, and there is no way of denoting this except for a complete program.

So one solution (there are others) to this problem is to introduce a kind of "delimiter" for the action of control operators, dubbed "prompt" by Felleisen [Fel88], by analogy with the "prompts" in programming-language interpreter top-levels. Thus, we have the following modification to our language:

Definition 2 $(\lambda + C + \#)$ Term Set: To λ + C add another unary term-forming operator, #, neither binding nor creating variables. V are values.

Evaluation (denoted $\rhd_{C\#v(LR)}$) :

Program Contraction Rules:

$$\begin{array}{rcl} E[(\lambda x.b)V] & \rhd_1 & E[b[V/x]] \\ E_1[E_0[\mathcal{C}k.M]] & \rhd_1 & E_1[M[(\lambda x.\mathcal{A}E_0[x])/k]] \\ E_1[E_0[\mathcal{A}M]] & \rhd_1 & E_1[M] \\ E_1[E_0[\#V]] & \rhd_1 & E_1[E_0[V]] \end{array}$$

Evaluation Contexts (Call-by-Value, Left-to-Right):

$$\begin{array}{rcl} E_0 & = & [\,] \mid E_0(N) \mid V(E_0) \\ E_1 & = & [\,] \mid E_1[E_0[\#[\,]]] \\ E & = & E_1[E_0[\,]] \end{array}$$

This evaluator specifies that the "extent?" of action of a. control operator is delimited by the "prompt." Thus, code which is run "inside" **a. prompt cannot** abortively esit, and, in particular, when t hat code computes a concrete value, the evaluation proceeds as if it were purely functional - no side-effects can be observed.

Now, naturally, one might ask: is there a way to "jump over" the delimiters? This would lead us to invent. a. new control-operator, which could jump over the prompts. and grab a.11 of the evaluation context, to the top of the program. And after that, we migh t want to delimit the action of t his opera tor. too. And so on. Thus, we come to the hierarchy of Felleisen & Sitaram [FS90], which is expressed as a. set of definitions, on top of the operators $\mathcal{F}$, a nd prompt. using **side-effects.**

Danvy & Filinski's hierarchy is another way of doing this, and has the ***wonderful*** property that its semantics is ***not*** expressed via side-effects, but rather via iterated CPS-translation. Here, we present first the "standard" semantic interpreter for the language, and then, a version where η-redexes have been removed. Both interpreters are found in [DF90], and are reproduced here for completeness. Finally, we will work with the CPS-translation version of the semantics, which is omitted, as it, can be trivially inferred from the interpreter presented below.

Definition 3 (Danvy & Filinski's Hierarchy) Let the integer $m > 1$ be a fixed parameter for the rest of this paper. Moreover, stipulate that all control operators are of "height" less than m – 1.

Term Set: To the CBV X-calculus, add two new families of term-forming operators indexed syntactically by $n > 0$, $\mathcal{S}_n k.M$, which binds k in M, and $\#_n(M)$, which neither binds nor creates free variables. Define the "height" of a control operator as its numeric subscript.

Semantic Interpreter: Suppose that a closed program P contains operators of "height" $m - 2$ or less. Let $(0 \leq i < m)$:

$$\theta_i \equiv \lambda v \kappa_{i+1} \cdots \kappa_m . \kappa_{i+1} v \kappa_{i+2} \cdots \kappa_m$$

and $\theta_m \equiv \lambda x.x$.

Interpretation Function:

$$[\![x]\!]\rho\kappa_1 \cdots \kappa_m \equiv \theta_0(\rho(x))\kappa_1 \cdots \kappa_m$$
$$[\![\lambda x.M]\!]\rho\kappa_1 \cdots \kappa_m \equiv \theta_0(\lambda u.\lambda\kappa'_1 \cdots \kappa'_m.[\![M]\!](\rho[x \mapsto u])\kappa'_1 \cdots \kappa'_m)\kappa_1 \cdots \kappa_m$$
$$[\![M(N)]\!]\rho\kappa_1 \cdots \kappa_m \equiv [\![M]\!]\rho(\lambda m\kappa'_2 \cdots \kappa'_m.[\![N]\!]\rho(\lambda n\kappa''_2 \cdots \kappa''_m.m(n)\kappa_1\kappa''_2 \cdots \kappa''_m)\kappa'_2 \cdots \kappa'_m)\kappa_2 \cdots \kappa_m$$
$$[\![\mathcal{S}_n k.M]\!]\rho\kappa_1 \cdots \kappa_n\kappa_{n+1}\kappa_{n+2} \cdots \kappa_m \equiv [\![M]\!](\rho[k \mapsto p])\theta_1 \cdots \theta_n\kappa_{n+1}\kappa_{n+2} \cdots \kappa_m$$
$$(where\ p \equiv \lambda v\kappa'_1 \cdots \kappa'_n\kappa'_{n+1}\kappa'_{n+2} \cdots \kappa'_m.\theta_0 v\kappa_1 \cdots \kappa_n(\lambda w\kappa''_{n+2} \cdots \kappa''_m.\theta_0 w\kappa'_1 \cdots \kappa'_n\kappa'_{n+1}\kappa''_{n+2} \cdots \kappa''_m)\kappa'_{n+2} \cdots \kappa'_m)$$
$$[\![\#_n(M)]\!]\rho\kappa_1 \cdots \kappa_n\kappa_{n+1}\kappa_{n+2} \cdots \kappa_m \equiv [\![M]\!]\rho\theta_1 \cdots \theta_n(\lambda v\kappa'_{n+2} \cdots \kappa'_m.\theta_0 v\kappa_1 \cdots \kappa_n\kappa_{n+1}\kappa'_{n+2} \cdots \kappa'_m)\kappa_{n+2} \cdots \kappa_m$$

Interpretation of Programs: The interpretation of P is (recall $\theta_m \equiv \lambda x.x$):

$$[\#_{m-1}(P)]\rho_{init}\theta_1 \cdots \theta_{m-1}\theta_m .$$

The stipulation that. a.11 control-operators have height less than $m - 1$ allows us to safely enclose the entire program in a prompt of height $m - 1$. This allows us to not concern ourselves with the interaction of control-operators and the "top" of the program.

This a.11 looks rather complicated, and perhaps *too* complicated. But there are several simplifications we can make. First, we can note that there are many η-redexes to be eliminated. Removing these yields:

$$\theta_i \equiv \lambda v\kappa.\kappa(v)\ (i < m)$$
$$\theta_m \equiv \lambda x.x$$
$$[\![x]\!]\rho\kappa_1 \equiv \theta_0(\rho(x))\kappa_1$$
$$[\![\lambda x.M]\!]\rho\kappa_1 \equiv \theta_0(\lambda u.[\![M]\!](\rho[x \mapsto u]))\kappa_1$$
$$[\![M(N)]\!]\rho\kappa_1 \equiv [\![M]\!]\rho(\lambda m.[\![N]\!]\rho(\lambda n.m(n)\kappa_1))$$
$$[\![\mathcal{S}_n k.M]\!]\rho\kappa_1 \cdots \kappa_n \equiv [\![M]\!](\rho[k \mapsto p])\theta_1 \cdots \theta_n$$
$$(where\ p \equiv \lambda v\kappa'_1 \cdots \kappa'_n\kappa'_{n+1}.\theta_0 v\kappa_1 \cdots \kappa_n(\lambda w.\theta_0 w\kappa'_1 \cdots \kappa'_n\kappa'_{n+1}))$$
$$[\![\#_n(M)]\!]\rho\kappa_1 \cdots \kappa_n\kappa_{n+1} \equiv [\![M]\!]\rho\theta_1 \cdots \theta_n(\lambda v.\theta_0 v\kappa_1 \cdots \kappa_n\kappa_{n+1})$$

Nest, we can notice that the semantics of shift is rather complex, and seems to contain instances of reset. So. we are lead to simplify shift, which yields a. new control operator. $\mathcal{C}_n$. the hierarchy's counterpart of Felleisen's $\mathcal{C}$:

$$[\![\mathcal{C}_n k.M]\!]\rho\kappa_1 \cdots \kappa_n \equiv [\![M]\!](\rho[k \mapsto \lambda v\kappa'_1 \cdots \kappa'_n.\theta_0 v\kappa_1 \cdots \kappa_n])\theta_1 \cdots \theta_n$$

The two operators, $\mathcal{S}_n$ and C,, are interdefinable in the presence of reset:

$$\begin{aligned} \mathcal{S}_n c.M &\equiv \mathcal{C}_n k.M[\lambda v.\#_n(kv)/c] \\ \mathcal{C}_n k.M &\equiv \mathcal{S}_n c.M[\lambda v.\mathcal{S}_n\_.(cv)/k] \end{aligned}$$

Indeed, the way to discover $\mathcal{C}_n$ is to expand the definition of $\mathcal{S}_n c.M[\lambda v.\mathcal{S}_n\_.(cv)/k]$, and perform all possible reductions. However, we are still a long way from a simple understanding of this semantics. We are still at the stage of reasoning only about CPS-translated programs. To have a better understanding, we would like:

- A C-rewriting machine evaluator, in the style of Felleisen and Friedman
- A set of local rewrite rules, which characterize evaluation
- A type system, which guarantees type-safety

We will find these things in the opposite order, discovering the type system by an analysis of typings of CPS-translated programs, and using typings to help us discover reductions, and so on. Let us, for the record, state what we wish to find:

> We wish to find a type system for source programs of the hierarchy, such that, under CPS-translation, well-typed source-programs always yield well-typed result-programs.

2.1 An Informal **Operational** Semantics

To help understand the terms of the hierarchy, it is useful to. have some sort of informal notion of how they compute. In the end, we will find that this informal notion coincides with the forma.1 one we develop, but *a priori*, we can develop some sort of very general idea of how programs execute. Intuitively, we think of a. program (in which all control operators are of "height" less than $m - 1$) as being either *values*, or composed of an *evaluation context*, within which resides a *term* of one of the following forms:

$$(\lambda x.M)V \quad \#_n(V) \quad \mathcal{S}_n k.M$$

In the first case, we have a. simple &contraction. In the second, the term is replaced by V. In the third, the evaluatiou contest may be devoid of reset's of height greater than $n - 1$. If this is the case, then the evaluation contest $E[\,]$ is wrapped in $\lambda x.\#_n(E[x])$, k is bound to this expression, and the current evaluation contest is set empty. If there is a. reset of height n or greater, then the evaluation contest up to that point is removed, and wrapped up as before, but that part of the evaluation contest comprising the reset, and that code outside it, is preserved.

Now, such a. description is quite informal, but essentially parallels the explanation fou nd in [DF90]. One of our tasks **(the** easiest) will be to verify that this description of evaluation is indeed correct. To begin with, we can write it **out** a little more explicitly:

Definition **4** ("Intuitive" C-Rewriting Machine) Evaluation Contexts:

$$\begin{aligned} C^{<n} &\equiv [\,] \mid C^{<n}(N) \mid V(C^{<n}) \mid \#_{<n}(C^{<n}) \\ C^{any} &\equiv [\,] \mid C^{any}(N) \mid V(C^{any}) \mid \#_i(C^{any}) \\ C^{\geq n} &\equiv [\,] \mid C^{any}[\#_{\geq n}([\,])] \end{aligned}$$

- $\#_{<n}([\,])$ **is** an instance $\#_i([\,])$, where $i <$ IL. Similarly with $\#_{\geq n}([\,])$

- $C^{<n}$ is a context which contains no prompts of level n or greater.
- C^{any} is a context which contains prompts of any level.
- $C^{\geq n}$ is either an empty context, or can contain anything, as long as its innermost frame is a prompt of level at least n.

Transitions:

$$\begin{array}{rcll} C^{any}[(\lambda x.B)V] & \triangleright_1 & C^{any}[B[V/x]] & (\beta_v) \\ C^{\geq n}[C^{<n}[\mathcal{S}_n k.M]] & \triangleright_1 & C^{\geq n}[M[\lambda x.\#_n(C^{<n}[x])/k]] & (\mathcal{S}) \\ C^{any}[\#_n(V)] & \triangleright_1 & C^{any}[V] & (\#) \end{array}$$

3 Pseudo-Classical Typing of C

The first task we must complete is to discover a type system for programs in the hierarchy, such that well-typed programs, under CPS-translation, yield well-typed purely functional programs. As a warm-up, we will show the equivalent development, for the call-by-value language $\lambda+\mathcal{C}$, yielding a pseudo-classical type system. We have already given the operational semantics of the call-by-value version of this language. We can also give a continuation semantics,

$$\begin{array}{rcl} [\![x]\!]\rho\kappa & \equiv & \kappa(\rho(x)) \\ [\![\lambda x.M]\!]\rho\kappa & \equiv & \kappa(\lambda u.\lambda\kappa'.[\![M]\!](\rho[x \mapsto u])\kappa') \\ [\![M(N)]\!]\rho\kappa & \equiv & [\![M]\!]\rho(\lambda m.[\![N]\!]\rho(\lambda n.m(n)\kappa)) \\ [\![\mathcal{C}k.M]\!]\rho\kappa_0 & \equiv & [\![M]\!](\rho[k \mapsto \lambda v.\lambda\kappa'.\kappa_0(v)])(\lambda x.x) \\ [\![\mathcal{A}M]\!]\rho\kappa & \equiv & [\![M]\!]\rho(\lambda x.x) \end{array}$$

which is provably equivalent to the call-by-value (left-to-right) operational semantics. (By partial evaluation, we could find a continuation-passing-style (CPS) translation, but this would be a waste of space, as it is evident.) Nest, we would like to discover a type system for this language. In a certain sense, if we wish a type system which directly speaks about the process of evaluation, then, since the contraction rules for control-operators manipulate the entire program, we must expect that the type system will somehow at least mention the type of the entire program. And this is exactly what happens:

Definition 5. (Pseudo-Classical Typing)

$$\frac{\Gamma, k : P \Rightarrow \bot \vdash_A M : \bot}{\Gamma \vdash_A \mathcal{C}k.M : P}\ \neg\neg\text{-}E \qquad \frac{\Gamma \vdash_A M : \bot}{\Gamma \vdash_A M : A}\ \bot\text{-}\top$$

$$\frac{\Gamma \vdash_A M : A}{\Gamma \vdash_A \mathcal{A}M : \bot}\ abort_1 \qquad \frac{\Gamma \vdash_A M : \bot}{\Gamma \vdash_A \mathcal{A}M : T}\ abort_2$$

$$\frac{\Gamma \vdash M : B \Rightarrow C \quad \Gamma \vdash_A N : B}{\Gamma \vdash_A M(N) : C}\ \Rightarrow\text{-}E$$

$$\frac{\Gamma, x : B \vdash_A M : C}{\Gamma \vdash_A \lambda x.M : B \Rightarrow C}\ \Rightarrow\text{-}I \qquad \frac{}{\Gamma, x : T \vdash_A x : T}\ ID$$

The intended meaning of a sequent $\Gamma \vdash_A M : T$ is that under the typing assumptions Γ, in a. complete program of type A, espression M has type T. Complete programs always have type $\vdash_T M : T$.

The important theorem we can prove is that this typing, under CPS-translation, yields an appropriately typed t ranslated term:

Definition 6 (Double-Negation/A-Translation) For ϕ an atomic type, define $\neg_{\phi}(T) \equiv T \Rightarrow \phi$.
CBV Translation on Types:

$$\begin{array}{rcll} \overline{\bot}^{\phi} & \equiv & \phi & \\ \overline{A}^{\phi} & \equiv & A & (A\ atomic) \\ \overline{A \Rightarrow B}^{\phi} & \equiv & \overline{A}^{\phi} \Rightarrow \neg_{\phi}\neg_{\phi}(\overline{B}^{\phi}) & \end{array}$$

CBV Translation on Sequents:

$$\begin{array}{c} x_1 : T_1, \cdots, x_n : T_n \vdash_{\phi} M : T \\ \longmapsto \\ x_1 : \overline{T_1}^{\phi}, \cdots, x_n : \overline{T_n}^{\phi} \vdash \underline{M} : \neg_{\phi}\neg_{\phi}(\overline{T}^{\phi}) \end{array}$$

Theorem 7 (Type-Translation)
If $\Gamma \vdash_{\phi} M : T$, then $\overline{\Gamma}^{\phi}$ t- $\underline{M} : \neg_{\phi}\neg_{\phi}(\overline{T}^{\phi})$.

Proof: By induction on the typing proofs. □

In addition, the evaluator enjoys subject-reduction in this type system.

Theorem 8 (Subject Reduction) Every program contraction rule of $\triangleright_{\mathcal{CV}(LR)}$, preserves pseudo-classical typing of programs $\vdash_T M : T$.

Proof: Mechanical checking. The only interesting cases are the program-contraction rules for A and $\mathcal{C}$.
Case $\mathcal{C}$: Since $E[\mathcal{C}k.M] \triangleright_1 M[(\lambda x.\mathcal{A}E[x])/k]$, we have the following typings:

$$\dfrac{\dfrac{k : T \Rightarrow \bot \vdash_{\phi} M : \bot}{k : T \Rightarrow \bot \vdash_{\phi} M : \bot} \quad \dfrac{\dfrac{\dfrac{\overline{x : T \vdash_{\phi} E[x] : \phi}}{x : T \vdash_{\phi} \mathcal{A}E[x] : \bot}\ abort_1}{\vdash_{\phi} \lambda x.\mathcal{A}E[x] : \neg(T)} \Rightarrow -I}{k : T \Rightarrow \bot \vdash_{\phi} M : \bot} \Rightarrow -E}{\vdash_{\phi} M(\lambda x.\mathcal{A}E[x]) : \phi}\ \textbf{I - T}$$

Case $\mathcal{A}$: Cases, depending on whether the control-string is typed with $abort_1$ or $abort_2$. In both cases, the contraction step is $E[\mathcal{A}M] \triangleright M$.
Case $abort_1$: Since the typing rule is $abort_1$, it follows that $M : \phi$.
Case $abort_2$: Since the typing rule is $abort_2$, it. follows that M : J-; hence, $M : \phi$ by **J--T.** □

In a. like manner, Felleisen's equational theories of control [FFKD86] also enjoy subject reduction in this type system.

3.1 Extended to Prompt/Reset

To extend this development to the operator reset, we simply examine the CPS-translation of reset:

$$\underline{\#(M)} \equiv \lambda\kappa.\kappa([\![M]\!]\rho(\lambda x.x))$$

Let us assume that we wish to produce a typing for the source terms, by analysis of the typings of the result terms. Further, suppose that the term $\underline{M}$ has type $\neg\neg_{\phi\,\phi}(T)$, where ϕ, T are atomic. Then $\lambda x.x$ will have type $T \Rightarrow \phi$ (hence $\phi \equiv T$). Thus, we can infer that $\vdash_T M : T$ will be the typing judgment for M (like a "top" of program - which is what a "prompt" is for). Next, we note that $\underline{M}(\lambda x.x)$ will have type T; hence $\underline{\#(M)}$ will have type $\neg\neg_{\psi\,\psi}(T)$, for any ψ. Thus, $\vdash_\psi \#(M) : T$ is (one) proper typing of the entire term. But if M contains free variables, then these could perhaps only be well-typed when the type under the turnstile was T. Hence, we must restrict Γ to contain only concrete types, since these will be invariant under translation:

Definition **9** (Pseudo-Classical Typing of Reset)

$$\frac{\Gamma \vdash_T M : T}{\Gamma \vdash_\psi \#(M) : T}\ \textit{reset}\ (\Gamma\ \textit{concrete})$$

We could also prove type-translation and subject reduction theorems, but since this is only the first stage of the hierarchy, we delay them until later.

3.2 An "Effect" Version of the Type System

Intuitively, the type under the turnstile in a judgment $\Gamma \vdash_\phi M : T$ is the type of the entire program in which M is embedded. Thus, any abortive terms in M must abort with expressions of this type, in order to have type-safety. In a case like $\Gamma, f : A \to B \vdash_\phi \#(M) : T$, if we allowed M to use f, then 'the sub-sequent would be $\Gamma, f : A \to B \vdash_T M : T$. But this says that if f, when applied to a value of type A, can abort with an espression of type ϕ, then it is well-typed in a contest where aborts should be with espressions of type T. Clearly, type-safety has been lost; hence the restriction that the hypothesis list should be concrete.

But perhaps there will be times when we actually do want this extra expressive power - to construct a function which aborts with an espression of type T, and then pass it to a. place where the function cannot be used in a well-typed manner, which will in turn pass it onwards, to a. place where it can be used. To give typings to these situations, we can always type CPS-terms directly. But this, as always, is cumbersome and awkward.

Instead, we can note that the call-by-value CPS-translation we have been using wraps a. $\neg\neg_{\phi\,\phi}(\bullet)$ (a "dou ble-ϕ-ation") around the conclusion, and around the righ t- hand-side of every fu nc tion- type in the program. This looks very much like a *computational effect*, and so we might. try to build an "effect" typing system, where we annotate function-types, and the type of the conclusion:

Definition 10 ("Effect" Version of Pseudo-Classical Typing) For ϕ an atomic type, define $\mathcal{K}_\phi[T] \equiv \neg\neg_{\phi\,\phi}(T)$.

$$\frac{\Gamma, k : P \Rightarrow \mathcal{K}_\phi[\phi] \vdash M : \mathcal{K}_\phi[\phi]}{\Gamma \vdash \mathcal{C}k.M : P}\ \neg\neg\text{-}E$$

$$\frac{\Gamma \vdash M : \mathcal{K}_A[A]}{\Gamma \vdash \mathcal{A}M : \mathcal{K}_A[\phi]}\ abort_1 \qquad \frac{\Gamma \vdash M : \mathcal{K}_A[\phi]}{\Gamma \vdash \mathcal{A}M : \mathcal{K}_A[T]}\ abort_2$$

$$\frac{\Gamma \vdash M : \mathcal{K}_A[B \Rightarrow \mathcal{K}_A[C]] \quad \Gamma \vdash N : \mathcal{K}_A[B]}{\Gamma \vdash M(N) : \mathcal{K}_A[C]}\ \Rightarrow\text{-}E$$

$$\frac{\Gamma, x : B \vdash M : \mathcal{K}_A[C]}{\Gamma \vdash \lambda x.M : \mathcal{K}_A[B \Rightarrow \mathcal{K}_A[C]]}\ \Rightarrow\text{-}I \qquad \frac{}{\Gamma, x : T \vdash x : \mathcal{K}_A[T]}\ ID$$

$$\frac{\Gamma \vdash M : \mathcal{K}_T[T]}{\Gamma \vdash \#(M) : \mathcal{K}_\psi[T]}\ reset$$

We can prove that this typing is simply a "wrapping/hiding" of the CPS-typing; note that CPS-translating the conclusion of a control-effect typed proof yields a valid typing; that is, if Γ I- $M : T$ in the effect-typing, then

$$\Gamma \vdash \underline{M} : T$$

also holds (when the "effect''-definitions have been unfolded, of course). Now, we have no restriction on the types of hypotheses in the typing of reset. However, we have an increase in the complexity of the types. This is to be expected, though, since possibly every hypothesis of functional type could be well-typed in a ***different*** context. Another way to look at this typing is as a close relative to the monad of continuations [Mog91].

It is once again not too difficult to show that this alternative typing is sound with respect to evaluation, as well as Felleisen's reduction rules. Again, we omit these proofs, as this system is a fragment of the hierarchy we will now consider.

The treatment of $\bot$ in this last version is different from that of our original pseudo-classical typing - in particular, the "effect" typing of C does not mention 1. This can be ameliorated, at the expense of some extensionality arguments (essentially, showing that $\underline{M} : (\mathrm{I} \to \phi) \to \phi$ implies $\underline{M} : (\phi \to \phi) \to \phi$, which follows by extensionality, and noticing that anything of type $\phi \to \phi$ is also of type $\bot \to \phi$).

We make one last comment about the rule reset. Notice that ϕ is free in the conclusion, and instead of $\mathcal{K}_\psi[T]$, we could have written $\bigcap_\psi \mathcal{K}_\psi[T]$, taken to mean the intersection over a.11 ψ of $\mathcal{K}_\psi[T]$. This type cannot even be expressed in the original classical logic; nevertheless, it expresses the intuitive meaning of a reset.

4 Pseudo-Classical Typing of the Hierarchy

At this point, we are ready to give a typing for the hierarchy, using the same mechanisms we used before.

4.1 Intuition

Before we give the type system, let's try to get some intuition for what is going on. In the case of $\lambda + \mathcal{C}$, we found that the process of CPS-translation induced a double-negation/A-translation on typings of programs. In particular, if $M : \mathrm{A} \to B$, and $N : A$, yielding $M(N) : B$, then (for any ϕ):

$$\begin{aligned}
\underline{M} &: \neg_\phi\neg_\phi(\overline{A}^\phi \to \neg_\phi\neg_\phi(\overline{B}^\phi)) \\
\underline{N} &: \neg_\phi\neg_\phi(\overline{A}^\phi) \\
\underline{M(N)} &\equiv \lambda k.\underline{M}(\lambda m.\underline{N}(\lambda n.m(n)k)) \\
\underline{M(N)} &: \neg_\phi\neg_\phi(\overline{B}^\phi)
\end{aligned}$$

Since the CPS-translations of applications in the hierarchy q-reduce to the same term, our first attempt to give translations to terms in the hierarchy would be to simply duplicate the work for $\lambda + \mathcal{C}$. But if we consider for a moment terms which have *not* been q-reduced. then we can see

immediately that a simple double-negation/A-translation does *not* suffice. Suppose that $m = 2$:

$$\begin{array}{rcl} [\![M(N)]\!] & \equiv & \lambda\kappa_1\kappa_2.[\![M]\!](\lambda m\kappa_2'.[\![N]\!](\lambda n\kappa_2''.m(n)\kappa_1\kappa_2'')\kappa_2')\kappa_2 \\ [\![M]\!] & : & (((a \to b \to c \to d) \to e \to f) \to g \to h) \\ [\![N]\!] & : & ((a \to c \to d) \to e \to f) \\ [\![M(N)]\!] & : & b \to g \to h \end{array}$$

Moreover, when we look at the "top" of the program:

$$\begin{array}{rcl} [\![M]\!] & : & ((a \to (a \to b) \to b) \to (c \to c) \to d) \\ [\![M]\!](\lambda v\kappa.\kappa(v))(\lambda x.x) & : & d \end{array}$$

and at the semantics of a constant value,

$$\begin{array}{rcl} c & : & a \\ [\![c]\!] & \equiv & \lambda\kappa_1\kappa_2.\theta_0 c\kappa_1\kappa_2 \\ & : & (a \to b \to c) \to b \to c \end{array}$$

we find that

- κ_1 expects to be passed a value, and something of the same type as κ_2.
- The result type of κ_1 is the same as that of κ_2.

If we repeat this for $m = 3$, we find:

- κ_1 expects to be passed a. value, and two other arguments, of the same types as κ_2 and κ_3.
- κ_2 expects to be passed a value, and something of the same type as κ_3.
- The result types of all three continuations are the same.

Thus, in the general case, where we have a program phrase, M, applied to 772 continuations, $[\![M]\!]\kappa_1 \cdots \kappa_m$, we find that each continuation κ_i expects to be applied to a value, and then to arguments of the types of each of the continuations $\kappa_{i+1} \cdots \kappa_m$, and finally, to return a. final answer.

In other words, suppose $[\![M]\!]$ is applied to $\kappa_1 \ldots \kappa_m$, respectively of types $\tau_1 \ldots \tau_m$, and produces a result of type α. Then each τ_i takes as arguments a value of some type, and values of types $\tau_{i+1} \cdots \tau_m$, and produces a. result of type α. So, the type of $[\![M]\!]$ can be written:

$$(T \to (\tau_2 \cdots \tau_m \to a)) \to \tau_2 \cdots \tau_m \to \alpha$$

This should immediately remind the reader of double-negation/A-translation, and indeed, if we look at the typing of application, with these remarks in mind, we find that the typing above, generated by ML, can be simplified, to

$$\begin{array}{rcl} [\![M(N)]\!] & \equiv & \lambda\kappa_1\kappa_2.[\![M]\!](\lambda m\kappa_2'.[\![N]\!](\lambda n\kappa_2''.m(n)\kappa_1\kappa_2'')\kappa_2')\kappa_2 \\ [\![M]\!] & : & ((n \to ((b \to o') \to o')) \to o') \to o' \\ [\![N]\!] & : & (a \to 0') \to 0' \\ [\![M(N)]\!] & : & (b \to o') \to o' \end{array}$$

where $o' \equiv (c \to d) \to d$. It should be obvious now that what happened to application was that instead of performing an A-translation with some atomic type ϕ, we did it with a type $(c \to d) \to d$. If we were to experiment some more. with, say, $m = 3$, we would find that d was in turn replaced by some dou ble-negated/A-translated type. The sizes of these types explode exponentially wit h m; nevertheless, the amount of information used to construct them is linear, and the construction is systematic; hence, we can invent some notation to help us wri te down the types:

Notation 11 (Iterated A-Translated Types) Define some abbreviations for sequences of type expressions/variables, and then the operator $\mathcal{K}$, which summarizes an iterated double-negation/A-translation.

$\overline{\alpha}$	:	a sequence of type-expressions, of fixed, but unspecified, length.
β^n	:	sequence of n identical type-expessions, β.
$\overline{\alpha^n}$	:	sequence of n different type-expressions.

$$\begin{aligned}\mathcal{K}_{[]}[T] &\equiv T\\ \mathcal{K}_{\overline{\alpha}\tau}[T] &\equiv (T \to \mathcal{K}_{\overline{\alpha}}[\tau]) \to \mathcal{K}_{\overline{\alpha}}[\tau]\end{aligned}$$

And with this notation, we can rewrite the typing of application:

$$\begin{aligned}[\![M(N)]\!] &\equiv \lambda\kappa_1\kappa_2.[\![M]\!](\lambda m\kappa_2'.[\![N]\!](\lambda n\kappa_2''.m(n)\kappa_1\kappa_2'')\kappa_2')\kappa_2\\ [\![M]\!] &: \mathcal{K}_{\tau_2\tau_1}[a \to \mathcal{K}_{\tau_2\tau_1}[b]]\\ [\![N]\!] &: \mathcal{K}_{\tau_2\tau_1}[a]\\ [\![M(N)]\!] &: \mathcal{K}_{\tau_2\tau_1}[b]\end{aligned}$$

4.2 The Type System

Finally, we have a.11 the tools we need to write down a type system for the hierarchy. Before we do so, we ought to say a word or two about intersection types. We will use the notation $\bigcap_{\alpha} T[\alpha]$ for the intersection over a.11 monomorphic types α, of $T[\alpha]$. We will only instantiate such intersections with concrete types (that is, free of implications). In the rule for intersection-elimination, we identify one subscripted type, β, for elimination. There might be other types which are quantified also, e.g. we might eliminate β_2 in $\bigcap_{\beta_3\beta_2\beta_1} \mathcal{K}_{\beta_3\beta_2\beta_1}[T]$, without changing the quantification of β_3, β_1.

Definition 12 (Type System for the Hierarchy) The type system has two kinds of judgments: $M : T$ and T *concrete*. The first is a judgment of membership of a program in a type, and the second, of the concreteness of a type. Likewise, there are two kinds of assumptions, of the same form. We do not go into details of how to prove that a type is concrete; essentially, a type variable declared to be concrete is thus, as are conjunctions, disjunctions, and primitive concrete types.

$$\frac{}{\Gamma, x : T \vdash_{\mathcal{H}} x : \mathcal{K}_{\alpha_m\cdots\alpha_1}[T]}\ var$$

$$\frac{\Gamma, x : A \vdash_{\mathcal{H}} M : \mathcal{K}_{\alpha_m\cdots\alpha_1}[B]}{\Gamma \vdash_{\mathcal{H}} \lambda x.M : \mathcal{K}_{\alpha_m\cdots\alpha_1}[A \to \mathcal{K}_{\alpha_m\cdots\alpha_1}[B]]}\ lambda$$

$$\frac{\Gamma \vdash_{\mathcal{H}} M : \mathcal{K}_{\alpha_m\cdots\alpha_1}[A \to \mathcal{K}_{\alpha_m\cdots\alpha_1}[B]] \quad \vdash N : \mathcal{K}_{\alpha_m\cdots\alpha_1}[A]}{\Gamma \vdash_{\mathcal{H}} M(N) : \mathcal{K}_{\alpha_m\cdots\alpha_1}[B]}\ apply$$

$$\frac{\Gamma \vdash_{\mathcal{H}} M : \mathcal{K}_{\alpha_m\cdots\alpha_{n+1}\beta^n}[\beta] \quad \Gamma \vdash_{\mathcal{H}} \beta\ concrete}{\Gamma \vdash_{\mathcal{H}} \#_n(M) : \bigcap_{\delta_n\cdots\delta_1} \mathcal{K}_{\alpha_m\cdots\alpha_{n+1}\delta_n\cdots\delta_1}[\beta]}\ reset$$

$$\frac{\Gamma, \kappa : T \to \bigcap_{\gamma_n \cdots \gamma_1} \mathcal{K}_{\alpha_m \cdots \alpha_{n+1} \gamma_n \cdots \gamma_1}[\beta] \vdash_{\mathcal{H}} M : \mathcal{K}_{\alpha_m \cdots \alpha_{n+1} \beta^n}[\beta]}{\Gamma \vdash_{\mathcal{H}} \mathcal{S}_n \kappa.M : \mathcal{K}_{\alpha_m \cdots \alpha_{n+1} \beta \delta_{n-1} \cdots \delta_1}[T]} \; shift$$

$$\frac{\Gamma \vdash_{\mathcal{H}} B : concrete \quad \Gamma \vdash_{\mathcal{H}} M : \bigcap_{\beta} \mathcal{K}_{\alpha_m \cdots \alpha_{n+1} \beta \gamma_{n-1} \cdots \gamma_1}[T]}{\Gamma \vdash_{\mathcal{H}} M : \mathcal{K}_{\alpha_m \cdots \alpha_{n+1} B \gamma_{n-1} \cdots \gamma_1}[T]} \cap - E$$

$$\frac{\Gamma, \beta : concrete \vdash_{\mathcal{H}} M : \mathcal{K}_{\overline{\alpha} \beta \overline{\gamma}}[T]}{\Gamma \vdash_{\mathcal{H}} M : \bigcap_{\beta} \mathcal{K}_{\overline{\alpha} \beta \overline{\gamma}}[T]} \cap - I$$

Notice, that in $\cap - E$, B is a new type, which we substitute in for β, and not a type-variable.

And, again, the type system enjoys a type-translation theorem with respect to the CPS-translation. Since this proof is what actually says that our type system makes sense, we do it in rather great detail. First, a few useful lemmas:

Lemma **13** (K-X-Intros) A term $\lambda \kappa_n \ldots \kappa_m.M$ (n $<$ m) has type $\mathcal{K}_{\alpha_m \cdots \alpha_n}[T]$ iff, under the assumptions:

$$\begin{aligned} \kappa_n &: T \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\alpha_n] \\ &\vdots \\ \kappa_i &: \alpha_{i-1} \to \mathcal{K}_{\alpha_m \cdots \alpha_{i+1}}[\alpha_i] \\ &\vdots \\ \kappa_m &: \alpha_{m-1} \to \alpha_m, \end{aligned}$$

M has type α_m.

Proof: Direct calculation. First step is that $\mathcal{K}_{\alpha_m \cdots \alpha_n}[T] \equiv (T \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\alpha_n]) \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[a,]$. □

Lemma 14 ($\mathcal{K}$-λ-**Elims**) A term $M(N_n) \ldots (N_m)$ ($n < m$) has type α_m when:

$$\begin{aligned} M &: \mathcal{K}_{\alpha_m \cdots \alpha_n}[\alpha_{n-1}] \\ N_n &: \alpha_{n-1} \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\alpha_n] \\ &\vdots \\ N_i &: \alpha_{n-1} \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\alpha_n] \\ &\vdots \\ N_m &: \alpha_{m-1} \to \alpha_m \end{aligned}$$

Proof: Direct calculation, with some induction, and some tests at the extrema. □

Lemma **15** (θ_i Typing) θ_i ($i < m$) ($\equiv \lambda v \kappa.\kappa(v)$) has type $T \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1} T}[T]$.

Proof: The desired type unfolds to $T \to (\mathrm{T} \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[T]) \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[T]$. If $v : T$, $\kappa : T \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[T]$, then $\kappa(v)$ has the desired type. □

Lemma 16 (θ_i Application I) For any types $\alpha_m \cdots \alpha_n$, α_{n-1}, $(n < m)$, $\theta_{n-1}M(N_n)\cdots(N_m)$ has type α_m when:

$$M : \alpha_{n-1}$$
$$N_n : \alpha_{n-1} \to \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\alpha_n]$$

$$N_m : \alpha_{m-1} \to \alpha_m$$

Proof: Since n < m, $\theta_{n-1} = \lambda x\kappa.\kappa(x)$. Thus, $\theta_{n-1}MN_n$ has type $\mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\alpha_n]$. The conclusion follows by Lemma 14. cl

Corollary 17 (θ_i Application II) If $M : \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}\beta}[\beta]$, then $M\theta : \mathcal{K}_{\alpha_m \cdots \alpha_{n+1}}[\beta]$.

Proof: By Lemma 15. □

Theorem 18 (Type-Translation) If I? $\vdash_{\mathcal{H}}$ A4 : T then Γ I- $[\![M]\!]$: T (i.e. with all abbreviations expanded.)

Proof: By induction on proofs: **[Note: This proof is found in the expanded version of the report]** ·I

However, since we have neither a C-rewriting machine evaluator, nor a set of local rewrite rules, there is really nothing else we can prove. Type-translation guarantees that evaluating well-typed programs by first translating them, and then running the translated programs, will never produce type errors. But it says nothing about, more direct means of understanding the evaluation process. To do this, we need a C-rewriting machine, and a calculus.

5 Local Reduction Rules

In this section, we find a sound equational reasoning system for our language. In the case of λ + C, Felleisen took the CPS-translation/abstract machine as a guide, and produced various reduction rules, which he then verified. But this process is essentially one of guesswork, and for a CPS-translation as large as the hierarchy, it seemed intractably difficult. However, by observing that all the reduction rules should preserve pseudo-classical typing, we can "cut down" the search space, and thus discover the following reduction rules:

Definition 19 (Reduction Rules for the Hierarchy)

$$(\mathcal{S}_n\kappa.M)(N) \to_1 \mathcal{S}_1c_1.\mathcal{S}_2c_2.\cdots\mathcal{S}_nc_n.M[\lambda f.\#_n(c_n(\cdots\#_2(c_2(\#_1(c_1(fN))))))/\kappa] \quad \mathcal{S}_L$$
$$(V)(\mathcal{S}_n\kappa N) \to_1 \mathcal{S}_1c_1.\mathcal{S}_2c_2\cdots\mathcal{S}_nc_n.N[\lambda a.\#_n(c_n(\cdots\#_2(c_2(\#_1(c_1(Va))))))/\kappa] \quad \mathcal{S}_{Rv}$$

$$\#_i(\mathcal{S}_jk.M) \to_1 \#_i(M[\lambda x.x/k]) \quad \mathcal{S}_{<\#}\ j \le i$$
$$\#_i(\mathcal{S}_jk.M) \to_1 \mathcal{S}_jk.M \quad \mathcal{S}_{>\#}\ \mathbf{j} > \mathbf{i}$$

$$\#_i(\#_j(M)) \to_1 \#_{max(i,j)}(M) \quad \#\#$$
$$\#i(V) \to_1 V \quad \#_v$$
$$(\lambda x.M)(V) \to_1 M[V/x] \quad \beta_v$$

This ruleset enjoys soundness, both with respect to the CPS-translation, and with respect to the pseudo-classical type system. First! we need a. few lemmas:

Definition 20 (S-Telescoping) The reduction $\mathcal{S}\uparrow$, the telescoping of $\mathcal{S}$, is defined as:

$$\mathcal{S}_nk.M \to \mathcal{S}_1c_1.\mathcal{S}_2c_2.\cdots\mathcal{S}_nc_n.M[\kappa \mapsto \lambda x.\#_n(c_n(\cdots\#_2(c_2(\#_1(c_1(x))))))]$$

While we do not add this rule to our set of reductions, we might want to, since it in a sense is already present, in the rules $\mathcal{S}_L$ and $\mathcal{S}_{Rv}$. If we were to add S $\uparrow$, we could simplify the other two rules significantly - they would only have to account for the case n = 1. We do not do this, only because our chosen ruleset seems easier to standardize. However, we will feel free to use S $\uparrow$, as a simplifying device only, while pointing out how to remove it, in favor of $\mathcal{S}_L$, $\mathcal{S}_{Rv}$.

Lemma 21 ($\mathcal{S}$-η) The two terms $\mathcal{S}_n\kappa.M$ and $\mathcal{S}_n c.M[\kappa \mapsto \lambda x.\#_n(c\ s)]$ are denotationally equivalent.

Proof: Direct calculation. The essential idea is that an abstracted context is going to already contain a reset, so adding another one will not change anything. □

Lemma 22 (S-Telescope-Step) The two terms

$$\mathcal{S}_n c.M[\kappa \mapsto \lambda x.C[\#_n(cx)]] \qquad \mathcal{S}_{n-1}c_{n-1}.\mathcal{S}_n c_n.M[\kappa \mapsto \lambda x.C[\#_n(c_n\#_{n-1}(c_{n-1}x))]]$$

and (where C[] is an arbitrary term-context) are denotationally equivalent.

Proof: Again, direct calculation. The essential idea is that a n-level abstracted context can be replaced by a tower of its n – l-level component, and the remainder at level n, in any place where the n-level context was used. □

Lemma 23 (S-Telescoping: Soundness Under Translation) The reduction S $\uparrow$ is sound in the seman tics.

Proof: This one is now simple. Use the $\mathcal{S}$-η lemma to unfold $\mathcal{S}_n k.M$ into $\mathcal{S}_n c.M[\kappa \mapsto \lambda x.\#_n(cx)]$, and then use the telescope-step lemma to unfold the captured context, $\lambda x.\#_n(cx)$ into $\lambda x.\#_n(c_n(\cdots\#_2(c_2(\#_1(c_1(x))))))$. cl

Lemma 24 (S-Telescoping: Typing) The rule S $\uparrow$ enjoys subject-reduction.

Proof: We do this in detail. **[Note: This proof is found in the expanded version of the report]** ❑

Theorem 25 (Type-Soundness of Reductions) The reductions above are a.11 type-sound; if the left-hand-side is a. well-typed program, then the right-hand-side is also, and of the same type.

Proof: By cases on the reduction rules. Again, since this says a lot about our programming language, we do it in detail. **[Note: This proof is found in the expanded version of the report]** ❑

Theorem 26 (Semantic Soundness of Reductions) The reductions above are all semantically sound; in the semantic interpretation, the two sides of each reduction are equal.

Proof: Another proof by cases: **[Note: This proof is found in the expanded version of the report]** □

6 A C-Rewriting Machine Evaluator

In order to show that our reduction ruleset is reasonable, we must show a C-rewriting machine, to which it can be compared. Since such has not been published, we give one here. It is constructed in essentially the same way as Felleisen's machine for $\lambda+\mathcal{C}$ - by a concretization of the continuation semantics, followed by a process of simplifying the abstract machine, until it became a rewriting system on source-code programs. We produce the development below. To give credit where credit is due, before we worked out this means of arriving at the correct C-rewriting machine, Danvy had informed us of his work in this same direction. [Dan92].

We show the machine because:

- It exists, and provides justification for the informal operational semantics
- More importantly, it becomes clear that even though we have this machine, to discover reduction rules for a calculus is far from simple, and indeed having the machine around does not (surprisingly) help very much.

We begin with the continuation semantics, and construct a series of machines, mimicking faithfully the development of Felleisen & Friedman [FF86]:

- a CEK^+-machine (control-string, environment, and list of one or more continuation codes)-
- a CK^+-machine (replacing the environment manipulation with direct substitution)
- a CC+-machine (replacing the continuation codes with marked Sk-contexts)
- a CC-machine (collapsing the several continuation codes into only one)
- a "rigorous" C-rewriting machine in the standard style of Felleisen & Friedman, but with a more complex definition of evaluation context,
- finally, an "intuitive" C-rewriting machine, with the same transitions as the previous machine, but with a more natural definition of evaluation contexts.

Since this process is relatively systematic, and space-consuming, we simply give the CEK^+-machine, and the "rigorous" C-machine (the "intuitive" C-machine has already been given).

Definition 27 **(CEK^+-Machine)** The CEK^+-machine is an extension of Felleisen & Friedman's CEK-machine, with extra continuation-strings. The original three components retain their original form:

- a control-string, which is either empty (‡), or contains a term,
- an environment, which is either empty, or contains a mapping from variables to either pairs of (term,environment), or continuation-context (to be defined later)
- a continuation-string, which is a list of frames, representing a. com bination of the argument and control stacks.

We add $m - 1$ more continuation strings:

- the **second** continuation **string stores a. list of'** first-level continuation **strings.**

- the third continuation string stores a list of pairs, of first- and second-level strings
- the n-th continuation string stores a list of n − 1-tuples, of $1, \ldots, n$ − l-level continuation-strings.

Finally, a continuation-context is a pair of a natural number n, and a list of continuation-strings, of levels $1, \cdots, n$, written [n, $\kappa_1, \ldots \kappa_n$].

The transitions of the machine are given in two parts: the first part are those transitions which it shares with the CEK-machine. For these, the "extra" continuation-strings are not affected, and are simply passed along. Thus they are not displayed. For the other transitions, since they affect the extra continuation-strings, we display the complete machine-state:

$$\begin{array}{lll}
\langle x & ,\rho, & \kappa\rangle \longmapsto \langle \ddagger\ , \quad \emptyset \quad , \quad \kappa \texttt{ ret } \rho(x)\rangle \\
\langle \lambda x.M & ,\rho, & \kappa\rangle \longmapsto \langle \ddagger\ , \quad \emptyset \quad ,\kappa \text{ ret } (\lambda x.M, \rho)\rangle \\
\langle M(N) & ,\rho, & \kappa\rangle \longmapsto \langle M, \quad \rho \quad , \quad \kappa \texttt{ arg } (N, \rho)\rangle \\
\langle \ddagger & ,\emptyset, & \kappa \text{ arg } (N,\rho) \text{ ret } F) \longmapsto \langle N, \quad \rho \quad , \quad \kappa \text{ fun } F) \\
\langle \ddagger & ,\emptyset,\kappa \text{ fun } (\lambda x.M,\rho) \text{ ret } V) & \longmapsto \langle M,\rho[x \mapsto V], \quad \kappa\rangle
\end{array}$$

$$\langle \ddagger, \emptyset, \square_1 \cdots \square_{m-1}, \square_m \texttt{ ret } V\rangle \longmapsto V$$

$$\begin{array}{lll}
\langle \mathcal{S}_n k.M\ , & \rho & , \quad \kappa_1 \cdots \kappa_n, \kappa_{n+1} \cdots \kappa_m\rangle \\
\longmapsto \langle M & ,\rho[k \mapsto [n, \kappa_1, \cdots, \kappa_n]], & \square_1 \cdots \square_n, \kappa_{n+1} \cdots \kappa_m\rangle
\end{array}$$

$$\begin{array}{lll}
\langle \ddagger & , \quad \emptyset & , \quad \kappa_1' \texttt{ fun } [n, \kappa_1 \cdots \kappa_n] \texttt{ ret } V, \kappa_2' \cdots \kappa_n', \kappa_{n+1}' \cdots \kappa_m'\rangle \\
\longmapsto \langle \ddagger & , \quad \emptyset & , \quad \kappa_1 \texttt{ ret } V, \kappa_2 \cdots \kappa_n, \kappa_{n+1}' \texttt{ then } (\kappa_1' \cdots \kappa_n') \cdots \kappa_m'\rangle
\end{array}$$

$$\begin{array}{lll}
\langle \#_n(M), & P & , \quad \kappa_1 \cdots \kappa_n \kappa_{n+1} \kappa_{n+2} \ldots \kappa_m\rangle \\
\longmapsto \langle M & , \quad P & , \quad \square_1 \cdots \square_n, \kappa_{n+1} \text{ then } (\kappa_1 \cdots \kappa_n), \kappa_{n+2} \cdots \kappa_m\rangle
\end{array}$$

$$\begin{array}{lll}
\langle \ddagger & , \quad \emptyset & , \quad \square_1 \cdots \square_{i-2}, \square_{i-1} \texttt{ ret } V, \kappa_i \cdots \kappa_m\rangle \\
\longmapsto \langle \ddagger & , \quad \emptyset & , \quad \square_1 \cdots \square_{i-2}, \square_{i-1}, \kappa_i \texttt{ ret } V, \kappa_{i+1} \cdots \kappa_m\rangle
\end{array}$$

$$\begin{array}{lll}
\langle \ddagger & , \quad \emptyset & ,\square_1 \cdots \square_n, \kappa_{n+1}' \texttt{ then } (\kappa_1 \cdots \kappa_n) \texttt{ ret } V, \kappa_{n+2}' \cdots \kappa_m'\rangle \\
\longmapsto \langle \ddagger & , \quad \emptyset & , \quad \kappa_1 \texttt{ ret } V, \kappa_2 \cdots \kappa_n, \kappa_{n+1}', \kappa_{n+2}' \cdots \kappa_m\rangle
\end{array}$$

In the process of going from the CEK^+-machine to the "rigorous" C-machine, we end up encoding the continuation-contexts, [n, $\kappa_1 \ldots \kappa_n$], as a. λ-abstraction. But this is operationally sound, and causes no problems.

Definition 28 ("Rigorous" C-Rewriting Machine for the Hierarchy) Evaluation Contexts:

$$\begin{array}{rcl}
C^1 & \equiv & []^1 \mid C^1(N) \mid V(C^1) \\
C^n & \equiv & []^n \mid C^n[C^{n-1}[\cdots C^1[\#_{n-1}([])]\cdots]]
\end{array}$$

Transitions:

$$\begin{array}{rcll}
C^m[\cdots C^1[(\lambda x.B)V]\cdots] & \triangleright_1 & C^m[\cdots C^1[B[V/x]]\cdots] & (\beta_c) \\
C^m[\cdots C^{n+1}[C^n[C^{n-1}[\cdots C^1[\mathcal{S}_n k.M]\cdots]]]\cdots] & \triangleright_1 & C^m[\cdots C^{n+1}[M[\lambda x.\#_n(C^n[C^{n-1}[\cdots C^1[x]\cdots]])/k]]\cdots] & (\mathcal{S}) \\
C^m[\cdots C^{n+1}[C^n[C^{n-1}[\cdots C^1[\#_n(V)]\cdots]]]\cdots] & \triangleright_1 & C^m[\cdots C^{n+1}[C^n[C^{n-1}[\cdots C^1[V]\cdots]]]\cdots] & (\#)
\end{array}$$

Lemma **29** (Equivalence of C-Machines) The "intuitive" and "rigorous" C-machines are identical rewriting machines.

Proof: Since every program is either a value, or outermost a prompt of level m − 1, it follows that every "rigorous" C-machine context "stack", $C^m[\cdots C^{n+1}[C^n[C^{n-1}[\cdot\,.\,.C^1[\,]\cdot\,.\,\cdot]]]\cdots]$ divides naturally into an "intuitive" C-machine context, $C^{\geq n}[C^{<n}[\,]]$, where $C^n[C^{n-1}[\cdot\,.\,\cdot C^1[\,]\cdot\,.\,\cdot]] = C^{<n}[\,]$. Also, the stack is already a $C^{any}[\,]$. Since both definitions are unique - given a program phrase, there is only one way of producing either, and since the transition rules are identical, it follows that the two machines are identical. □

We delay proofs of subject reduction for the machines, since the representation theorems (that we can represent the course of computation of the C-rewriting machines by series of well-typed, sound, reduction rules), will immediately imply subject reduction.

Lemma **30** (Well-Typed Programs are not Wrong) Well typed programs in the hierarchy are either values, or contain C-machine redexes.

Proof: This proof is rather trivial, since every possible term in the hierarchy is either a value or a redex. We have not included constants; hence there are no cases of *wrong*. Nevertheless, even in the presence of function and basic constants, the proof still goes through. □

Theorem **31** (Semantic Equivalence) The semantic interpreter and the "intuitive" C-machine compute the same results when restricted to programs of concrete type. Formally, the $[\![M]\!]\theta_1\,.\,.\,\cdot\theta_m$ evaluates to some concrete value b, if and only if the intuitive C-machine evaluates the program M to the same value. This proof does not depend on the stipulation that all control-operators in the program have height less than $m - 1$, nor on the program being "wrapped" in a prompt of height $m - 1$.

Proof: We use the method of Felleisen & Friedman [FF86]. One shows that the CEK^+, CK^+, CC^+, and CC-machines all compute in lockstep. We then prove a unique context lemma, and show that one step of the CC-machine induces one or zero steps of the rigorous C-machine. Since the CC-machine is in lockstep with the semantics, this means that each step of computation semantics becomes zero, one, or two steps of the C-machine. (The case of two steps occurs when a continuation-context: coded as a λ-abstraction, is applied.) Termination of the semantic interpreter implies termination of the C-machine.

To get the other direction, we use the fact that the rigorous C-machine enjoys subject reduction) and that well-typed programs are either values, or have C-machine redeses. Hence, if the rigorous C-machine terminates, it will be with a value. This value maps back to a. final state of the CC-machine. Finally, the equivalence of the intuitive and rigorous C-machines finishes the job. □

6.1 Representation (Standardization/Correspondence?)

So now we know that the reductions preserve typing, and that they are semantically sound. a nd we have an evaluator. The logically next obligation is some sort of standardization/correspondence theorem. Jt seems intuitively clear that this ruleset contains enough reductions to standardize, for the same reasons that. Felleisen's ruleset was enough. However, just as with Felleisen's calculi, the ruleset does not contain enough reductions to prove a. direct correspouclence theorem, again. for the same reasons as for control-operator calculi - the process of lifting a control-operator out of a

context introduces many β-redexes, which are not contracted by the standard reduction sequence, and are not even β_v-redexes.

Thus, here, we will add a few other reductions, and prove a representation theorem. We make no attempt to prove a standardization theorem, even though that should be technically feasible, essentially since it would be a great waste of time and space. Even more of a waste would be a correspondence theorem, as it would be another (even more complex) rendering of Felleisen's analysis of encoded contexts. Nevertheless, three facts:

- the reductions we add are simple
- their use is very controlled
- The lack of these reductions was essentially what caused problems with a direct correspondence theorem for Felleisen's investigation of C (on the other hand, related reductions appear in work of Talcott [Tal])

lead us to believe that correspondence is a "mere" technical matter. The proofs in this section depend on the stipulation that all control-operators are of height less than $m - 1$, since this allows us to finesse issues of the "top" of a program. First, some definitions and technical details:

Definition 32 (Extra Reductions: β_E) Define the reductions β_E by:

$$\begin{aligned}
\#_1((\lambda x.\#_1(xN))M) &\rightarrow_1 \#_1(MN)\\
\#_1((\lambda x.\#_1(Vx))N) &\rightarrow_1 \#_1(VN)\\
\#_n((\lambda x.x)M) &\rightarrow_1 \#_n(M)\\
\#_n((\lambda x.\#_n(V_n\#_{n-1}(V_{n-1}\cdots\#_1(V_1x)\cdots)))\#_{n-1}(M)) &\rightarrow_1 \#_n(V_n\#_{n-1}(V_{n-1}\cdots\#_1(V_1\#_{n-1}(M))))
\end{aligned}$$

Lemma 33 (β_E Soundness) The β_E reductions are type-sound, as well as being sound under CPS- translation.

Proof: Direct calculation. □

Notation 34 (SS) The notation $\mathcal{S}\mathcal{S}_{i\ldots j}c_i\cdots c_j.M$ $(i \leq j)$ is shorthand for Sic;. . . $\cdot\mathcal{S}_jc_j.M$.

Theorem 35 (Representation) The ruleset already given, along with β_E, suffices to represent the "intuitive" C-rewriting machine exactly. By the correspondence between the intuitive and rigorous machines, we thus have representation of the rigorous machine also.

Proof: By cases on the program-contraction rules. **[Note: This proof is found in the expanded version of the report]** □

It is interesting that the reductions we discovered here were *not* simple generalizations of Felleisen's ruleset. Indeed, we first tried such simple generalizations, and found that they were type-unsound; hence we were forced to search further. Upon finding these rules, calculating their denotations told us that we had indeed succeeded; but without the pseudo-classical typing to guide us, it would have been a much longer process.

As can be seen from the proof, the reductions β_E are used in very specific places, to "clean up" the estra redexes produced by "lifting" a $\mathcal{S}$ **up** to its corresponding reset. We find four recurring patterns of "gar bage" , produce by lifting a. $\mathcal{S}$ from the left. and right, sides of application nodes, past a. lower-numbered reset, and finally, **when** we actually arrive at our destination. It seems clear that these patterns could be characterized, and a. correspondence **theorem such** as obtained by Felleisen & Friedman could be obtained **here also.**

7 An Example

For our example, we will type a program which decides if one list is a suffix of another. This program is surely taken from one of Danvy's publications, though we first learned of it directly from him [Dan91].

```
fun flip () ≡ S1c. (c tt) or (c ff)
fun rec suffix l ≡ if l = nil → nil
          I flip() = true → l
          | else → suffix(cdr l)
fun suffix-p l1 l2 ≡ #1(l1 = suffix l2)
```

So with a few definitions of primitive operators:

$$\frac{}{\textit{l-nil} : \bigcap_{\overline{\alpha}} \mathcal{K}_{\overline{\alpha}}[\textit{list}]}\ \textit{nil}$$

$$\frac{}{\text{t- tt , ff} : \bigcap_{\overline{\alpha}} \mathcal{K}_{\overline{\alpha}}[\textit{bool}]}\ \textit{truth}$$

$$\frac{}{\vdash \bullet = \bullet : \bigcap_{\overline{\alpha}} \mathcal{K}_{\overline{\alpha}}[T \to \mathcal{K}_{\overline{\alpha}}[T \to \mathcal{K}_{\overline{\alpha}}[\textit{bool}]]]}\ \textit{eq}$$

$$\frac{}{\vdash \bullet \text{ or } \bullet : \bigcap_{\overline{\alpha}} \mathcal{K}_{\overline{\alpha}}[\textit{bool} \to \mathcal{K}_{\overline{\alpha}}[\textit{bool} \to \mathcal{K}_{\overline{\alpha}}[\textit{bool}]]]}\ \textit{or}$$

$$\frac{\vdash B : \mathcal{K}_{\overline{\alpha}}[\textit{bool}] \quad \text{t-u} : \mathcal{K}_{\overline{\alpha}}[T] \quad \vdash V : \mathcal{K}_{\overline{\alpha}}[T]}{\vdash \textit{if}(B; U; V) : \mathcal{K}_{\overline{\alpha}}[T]}\ \textit{if}$$

$$\frac{f : A \to \mathcal{K}_{\overline{\alpha}}[B], x : A \vdash M : \mathcal{K}_{\overline{\alpha}}[B]}{\vdash \textit{fix}(f, x.M) : \mathcal{K}_{\overline{\alpha}}[A \to \mathcal{K}_{\overline{\alpha}}[B]]}\ \textit{rec}$$

we can type our program, and find that *suffix-p* has the type

$$\bigcap_{\overline{\alpha}} \mathcal{K}_{\overline{\alpha}}[\textit{list} \to \mathcal{K}_{\overline{\alpha}}[\textit{list} \to \mathcal{K}_{\overline{\alpha}}[\textit{bool}]]]$$

$\vdash$ flip : $\mathcal{K}_{bool}[unit \to \mathcal{K}_{bool}[bool]]$
  I- $\mathcal{S}_1$c. (c tt) or (c ff) : $\mathcal{K}_{bool}[bool]$
   c:$bool \to \bigcap_{\tau} \mathcal{K}_{\tau}[bool]$ t- (c tt) or (c ff) : $\mathcal{K}_{bool}[bool]$
      t- (c tt), (c ff): $\mathcal{K}_{bool}[bool]$

I- suffix:$\mathcal{K}_{bool}[list \to \mathcal{K}_{bool}[list]]$
 l:list $\vdash$ if(l=nil;nil;if(flip();l;cdr(l))):$\mathcal{K}_{bool}[list]$
    I- l=nil:$\mathcal{K}_{bool}[bool]$
    I- flip() : $\mathcal{K}_{bool}[bool]$
    $\vdash$ 1 : $\mathcal{K}_{bool}[list]$
    $\vdash$ cdr(l) :$\mathcal{K}_{bool}[list]$

$\vdash$ suffix-p : $\mathcal{K}_{\alpha}[list \to \mathcal{K}_{\alpha}[list \to \mathcal{K}_{\alpha}[bool]]]$
 l_1, l_2: $\vdash$ $\#_1(l_1 = \text{suffix } l_2)$: $\mathcal{K}_{\alpha}[bool]$
    $\vdash$ $(l_1 = \text{suffix } l_2)$: $\mathcal{K}_{bool}[bool]$
      suffix l_2 : $\mathcal{K}_{bool}[bool]$

8 Is This A-Translation?

In the literature, Friedman's A-translation [Fri78] is characterized in two ways. First, one can begin with a double-negation translation from classical to intuitionistic logic, and then, disjoin every atomic formula with a fresh constant, A. Other variants exist, for instance, disjoining every proposition (everywhere) with A. But another, and equivalent, way to do this is to stipulate that the double-negation translation is from classical to minimal logic, and then, the A-translation becomes the replacement of falsehood with some fresh constant A. [Lei85]

When we look at the relation with denotational semantics, we see that a double-negation translation to minimal logic, when considered along with a compatible translation on proofs, induces easily a denotational semantics of classical logic. (On the other hand, a double-negation translation into intuitionistic logic which is *not* into minimal logic also does *not* induce such a semantics - the Kuroda negative translation is such a counter-example).

In the setting, then, of double-negation translations into minimal logic, the choice of the A, is nothing more, and nothing less, than the choice of an *answer-type* for the denotational semantics.

Now all of this works well for simple double-negation translation, which corresponds with the standard CPS-translations, and maps classical logic into minimal logic. What happens when we consider the type system of the hierarchy, and the undeniably complex semantic interpreter? Quite simply, this semantic interpreter effects an *iterated double-negation/A-translation*. Consider the types of elements of an application, under a standard call-by-value double-negation/PI-translatioui and assume that U, V are atomic types:

$$\begin{array}{rcl} M(N) & : & V \\ M & : & U \to V \\ N & : & U \end{array} \quad \longmapsto \quad \begin{array}{rcl} \underline{M(N)} & : & \neg_{\phi}\neg_{\phi}(V) \\ \underline{M} & : & \neg_{\phi}\neg_{\phi}(U \to \neg_{\phi}\neg_{\phi}(V)) \\ \underline{N} & : & \neg_{\phi}\neg_{\phi}(U). \end{array}$$

So far, so good. The type ϕ is our fresh constant in the A-translation. Now, suppose we were to use the first level in the hierarchy to translate these programs. That is, we want the hierarchy's semantic interpreter, where $m = 2$. Then the types become:

$$\begin{array}{rcl} [\![M(N)]\!] & : & \mathcal{K}_{\phi_2\phi_1}[V] \\ [\![M]\!] & : & \mathcal{K}_{\phi_2\phi_1}[U \to \mathcal{K}_{\phi_2\phi_1}[V]] \\ [\![N]\!] & : & \mathcal{K}_{\phi_2\phi_1}[U] \end{array}$$

And if we unwind the definition of $\mathcal{K}_{\phi_2\phi_1}[\bullet]$, we find:

$$\mathcal{K}_{\phi_2\phi_1}[T] \equiv (T \to \mathcal{K}_{\phi_2}[\phi_1]) \to \mathcal{K}_{\phi_2}[\phi_1]$$

which is *exactly*

$$\neg_{\mathcal{K}_{\phi_2}[\phi_1]}\neg_{\mathcal{K}_{\phi_2}[\phi_1]}(T)$$

That is,

- Already we know that to get. the second level of the semantic interpreter from the standard CBV CPS-translation, it suffices to do some η-expansions.
- At the level of types, this corresponds to replacing ϕ by $\mathcal{K}_{\phi_2}[\phi_1]$ - *and that is all.*

Thus, we can see that the operators $\mathcal{K}$ are really iterated double-negation/A-translations. They hide this mess, since in reality the structure being computed over is much simpler - a linear-length stack of contexts, and not an exponential-size type.

Indeed, this is where the essential simplification in the type system for the hierarchy arises - in noticing that instead of replacing $\perp$ by ϕ in double-negation/A-translation, if we replace it by $\phi_2^{\phi_1}(\phi_1)$, we can:

- specify ϕ_1, which specifies a result-type for the first level of contexts,
- repeat the process, allowing us to give types for higher-level contexts,
- or terminate, by assigning a concrete type, which allows $\mathcal{C}$ to give the type of the entire program.

Thus, while our translation from the hierarchy's type system into a purely intuitionistic type system is *not* a standard double-negation/A-translation, we feel that it deserves the name A-translation, in the sense that, just as for the original A-translation, we are replacing an indeterminate answer-type in a typed denotational semantics, by a defined one, which can itself contain yet another indeterminate type, and so on.

9 Related Work

Obviously, our work is related to that of Danvy & Filinski [DF90], which it extends. Our typing also extends that of Danvy & Filinski [DF89] to a hierarchy, and also seems to simplify the meanings of various typings even for the first-order hierarchy case (though this has not been verified in detail). We use many techniques first invented by Felleisen, Friedman, and the team at Indiana. [FF86, FFKD86, Fel87], and also further researched by Felleisen and his team at Rice [Fel88, FS90]. 0 ur work on typing the hierarchy estends that of Griffin [Gri90], and is a. direct outgrowth of our work on program extraction from classical proofs [Mur91]. There has been other work on typing control-operators in higher-order functional/imperative languages: Harper et. al. [BD91] gave typings in monomorphic languages, and Harper and Lillibridge found that the standard methods of giving typings to control-operators did not work under erasure polymorphism [HL91]. 0 ne distinguishing feature of most work on typing control operators is that it has dealt with continuations as essentially single-ended objects - an input, but no output. When we consider the hierarchy, though, the standard way of abstracting a context produces a two-ended continuation - indeed, we must explicitly wrap the abstracted context in an abort in order to mimick the behaviour of $\mathcal{C}$. Thus, it becomes crucial to deal with the types of the *output-ends* of continuations. Another close relative of our work is that of Moggi and others on monads [Mog91] - in particular, there seem to be strong correspondences between our "wrappings" and the monad of continuations.

Our work on the reduction system appears to have close parallels in the computational lambda-calculus [Mog91], and Talcott's work on IOCC [Tal], in that their "extra" reductions, and ours, are motivated by the same concerns - recapturing the evaluator directly as a sequence of rewrite steps.

10 Conclusions and Directions for Future Work

This paper has focused on how. applying standard techniques, one can take a. new control-operator language, and discover the standard reasoning systems for it. We began with a. continuation

semantics for Danvy & Filinski's hierarchy of control operators, and found a type system for the language which enjoyed the appropriate relationship with the continuation semantics, viewed as a translation from the hierarchy into a pure functional language.

Next, we took the pseudo-classical type system, and, using it as a guide, discovered operationally sound observational congruences - enough of them to represent the evaluator, and of these, a subset which were intuitively enough to standardize. The importance of pseudo-classical typing here was, again, that it gave us a powerful tool in the search for valid equivalences - since checking equivalences by translation is difficult, being able to check them by typing them was a significant aid.

We then proceeded to develop a C-rewriting machine evaluator, using Felleisen & Friedman's methods of syntactifying abstract machines. Their method also renders the semantic equivalence proof routine, though tedious.

For future work, it seems that standardization and Church-Rosser theorems are de rigueur for the reduction system. Likewise, for the type system, extensions to polymorphism (perhaps not Milner-style polymorphism, of course!), recursive types, fixpoints, etc., seem tenable and necessary. Moreover, further work will be necessary to see if the reduction system proposed is indeed the ideal one. Unlike C, there is great room to maneuver in the construction of the calculus; for instance, we noted that instead of the complex rules $\mathcal{S}_L$, $\mathcal{S}_{Rv}$, we could have used the telescoping rule, S ↑, and first-level versions of the two lifting rules.

Another avenue of research is to experiment with a call-by-name version of this hierarchy. While this direction seems rather theoretically motivated, one interesting reason to do this would be to integrate control operators *directly* into lazy functional languages, rather than thru the standard uses of monads, which normally produce call-by-value control. On the other hand, a call-by-name version of the hierarchy could be coded into a lazy functional language, by using monad-like techniques. Even a "Reynolds"-interpreter - that is, a. language which admits either call-by-name or call-by-value functions, seems possible, and even easily achievable. Of course, the value of call-by-name control languages has y'et to be conclusively established.

What we find most striking about our entire development is that at every step along the way, we were lead by the type system. The type system gave us our first intuitions about the operational behaviour of terms, in a way which we could then reinforce by discovering reduction rules, and afterwards, a C-rewriting machine. Again, we discovered our reduction rules by searching for patterns which preserved typing.

Finally, perhaps the most important (and also the most intracta.ble) task is to extend this type system to encompass some of the patterns of usage of the hierarchy. We have seen that we can provide non-dependent typings for programs, but it would be even more satisfying to have total-correctness typings also. This seems intractable, since the hierarchy can be used to simulate assignable variables. Nevertheless, the pa&tern of usage of the hierarchy in programming suggests that such a. typing is indeed possible.

To sum up, we have attempted in this paper to explain how a new technique for giving type systems to control-operator languages - pulling back type systems thru denotational semantics - yields type systems for Danvy & Filinski's hierarchy. The obtained type system illuminates the operational behaviour of the hierarchy, aiding us in constructing the standard reasoning tools for understanding this programming language.

Acknowledgments

I thank Olivier Danvy and Andrzej Filinski, and .J ulia Lawall, for long discussions, electronic and personal, about t he hierarchy. Olivier also provided t lie CPS-converter which was used to verify

guesses for reduction rules. Thanks also go to Carolyn Talcott, for spotting many last-minute problems in presentation.

References

[BD91] D. MacQueen B. Duba, R. Harper. Typing first-class continuations in ML. In Conference Record *of the Eighteenth Annual ACM Symposium on Principles of Programming Languages,* pages 163–173, 1991.

[CR86] W. Clinger and J. Rees. The *revised*^{3} report on the algorithmic language scheme. *SIGPLAN Notices,* 21(12):37–79, 1986.

[Dan91] 0. Danvy. Personal communication, 1991.

[Dan92] 0. Danvy. Personal communication, 1992.

[DF89] 0. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical report, DIKU, 1989.

[DF90] 0. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Progmmming,* pages 151-160, 1990.

[Fel87] Matthias Felleisen. *The Calculi of λ_v-CS conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Progmmming Languages.* PhD thesis, Indiana. University, 1987.

[Fel88] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual A CM Symposium on Principles of Programming Lartguages,* pages 180-190, 1988.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD machine and the X-calculus. In *Formal Description of Programming Concepts* III, pages 131-141. North-Holland, 1986.

[FFKD86] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings, Symposium on Logic in Computer Science,* pages 131-141. IEEE Computer Society, 1986.

[Fri78] Harvey Friedman. Classically and intuitionistically provably recursive functions. In Scott, D. S. and Muller, G. H., editor, *Higher Set Theory,* volume *699* of *Lecture Notes in Mathematics,* pages 21-28. Springer-Verla.g, 1978.

[FS90] M. Felleisen and D. Sitaram. Control delimiters and their hierarchies. *Lisp and Symbolic Computation,* 3:67–99, 1990.

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 1990.*

[HL91] Harper, Robert and Lillibridge, Mark. ML with callcc is unsound, 1991. Electronic mail from types@theory.lcs.mit.edu.

[Lei85] D. Leiva.nt. Syntactic translations and provably recursive functions. *Journal Of Symbolic Logic,* 50(3):682–688, 1985.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation,* 93(1):55–92, July 1991.

[Mur91] Chet Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science,* 199 1.

[Rey72] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th A CM National Conference,* pages 7 I 7–740. ACM. 1972.

[Tal] C. Talcott. A theory for program and data type specifica tion. *Theoretical Computer Science.* To appear in the DISCO 1990 special issue.

On PaiLisp Continuation and its Implementation

Takayasu Ito and Tomohiro Seino
Department of Information Engineering
Faculty of Engineering
Tohoku University
Sendai, Japan 980

Abstract

PaiLisp continuation is an extension of Scheme continuation, introduced in a Scheme-based parallel Lisp language PaiLisp. Unlike Multilisp continuation an invocation of PaiLisp continuation can change the flow of control in another process, since for each process PaiLisp continuation packages up its process-id and continuation as a functional object. Actually the PaiLisp continuation can be used to kill a process. As in Scheme and Multilisp the PaiLisp continuations are denoted as (call/cc *f*) , where *f* must be a procedure of one argument. A continuation captured by call/cc may be invoked many times, but multiple-use continuations often incur some troublesome semantic problems in concurrent interactions between continuations and concurrency constructs. PaiLisp provides a new construct (`call/ep` *f*), a. restricted PaiLisp call/cc for the single-use continuation. Some details of PaiLisp call/cc and `call/ep` are explained, using several examples.

A PaiLisp interpreter is implemented on a shared memory parallel machine Alliant FX/80. Examples of using PaiLisp `call/cc` on this interpreter are given to demonstrate its behaviors, and some evaluations of PaiLisp call/cc are also given.

1 Introduction

PaiLisp continuation is an estension of Scheme continuation, introduced in a Scheme-based parallel Lisp language PaiLisp [ItoM90, Ito91]. Multilisp [Hals84, Hals90] is another Scheme-based parallel Lisp, and it adopts Scheme continuation in a straightforward manner. An invocation of Multilisp continuation closure does not give any effect to other processes. Unlike Multilisp continuation an invocation of a PaiLisp continuation can change the flow of control in another process, since for each process PaiLisp continuation packages up its process-id and continuation as a functional object. PaiLisp continuation is a natural extension of Scheme continua&ion into concurrency. The process continuation of Hieb-Dybvig [HieD90] also offers another interesting extension of continuation in to concurrency, but it is based on so-called $\mathcal{F}$-continuation of Felleisen et. al. [FeWFD88, Fell88]. Halstead's excellent. survey [Hals90] contains some nice comparative remarks on these continuations.

In this paper we discuss

1. PaiLisp continuation, call/cc and its restricted version `call/ep`

2. how to use PaiLisp call/cc and **call/ep**

3. an implementation of PaiLisp continuation and its evaluations

2 PaiLisp Continuation

PaiLisp continuation was introduced in a parallel Lisp language PaiLisp and its kernel PaiLisp-Kernel as an extension of Scheme continuation. PaiLisp-Kernel is a small subset of PaiLisp, defined to be Scheme with four concurrency constructs. PaiLisp continuation was demonstrated its usefulness in describing PaiLisp constructs by use of PaiLisp-Kernel. This was shown in [ItoM90]. See [ItoM90] and [Ito91] for more details of PaiLisp and PaiLisp-Kernel.

Before discussing on PaiLisp continuation we briefly explain about Scheme continuation and Multilisp continuation.

2.1 Scheme continuation

The notion of continuation was originally introduced in Denotational Semantics to model the rest of computation following from a point of the computation [Stoy81, Sch86]. Continuations are known to be powerful to denote semantics of various control structures in traditional programming languages. Scheme is a dialect of Lisp to provide continuation as a basic construct. In Scheme the continuations denoted as call-with-current-continuation are the procedures that programmers can manipulate directly as a first class object of Scheme. The Scheme procedure call-with-current-continuation is usually abbreviated as call/cc. It has the following syntax:

(call/cc *proc*)

where *proc* must be a procedure of one argument. A continuation created by call/cc packages up the current continuation as an escape procedure of one argument and passes it as an argument to *proc*. When the escape procedure is applied to its argument. its current continuation will be discarded. Instead the continuation that the escape procedure was created will become in effect. Intuitively speaking "continuation" behaves just like a control stack of an interpreter.

In the style of denotational semantics we may write as

$$\rho[\texttt{call/cc}] = \lambda\epsilon\kappa.\epsilon(\lambda\epsilon'\kappa'.\kappa\epsilon')\kappa$$

where ρ is an environment such that $\rho \in Env : \mathrm{I} \rightarrow Val$. This may be read as follows:

> call/cc receives a. procedure ϵ of one argument and a continuation κ, then apply ϵ to a continuation closure $(\lambda\epsilon'\kappa'.\kappa\epsilon')$. When this continuation closure $(\lambda\epsilon'\kappa'.\kappa\epsilon')$ is applied to ϵ' with the continuation κ' the continuation κ' will be discarded. and the continuation κ will be executed with ϵ', where κ is the continuation at the time of creating the con tin uation closure $(\lambda\epsilon'\kappa'.\kappa\epsilon')$.

2.2 Multilisp continuation

Multilisp adopted Scheme continuation into parallel Lisp in a straightforward manner. In case of Multilisp continuation the content of the control stack that executes call/cc will be kept in the continuation closure, and when its continuation closure is invoked the content of the control stack will be copied into the control stack of the process that invokes the continuation. The continuation captured by Multilisp call/cc will be executed by a process that invokes the continuation. Hence in case of Multilisp continuation there is no chance that an invocation of a continuation closure gives any effect to other processes.

Remark: Single-use and multiple-use continuations

A continuation captured by call/cc may be invoked many times. But, as is discussed in [Hals90] it is useful to distinguish between single-use and multiple-use continuations from the standpoint of efficiency and concurrent process interactions. According to our Scheme programming experiences Scheme continuations are used in the "single-use" style in most of actual Scheme programs. `call/ep` proposed by the first author is such a restricted call/cc based on the single-use continuation with the following syntactic definition:

(`call/ep` *proc*)

where *proc* must be a procedure of one argument as in (call/cc *proc*). An experimental sequential Scheme with this `call/ep` was implemented by 0. Hishida (a former student of the first author) [HisI88] and it was adopted and estended in PaiLisp [ItoM90].

2.3 PaiLisp continuation

PaiLisp continuation was introduced in the course of designing PaiLisp-Kernel. PaiLisp-Kernel is a small subset of PaiLisp, defined to be Scheme with four concurrency constructs; that is,

PaiLisp-Kernel = Scheme + {spawn, suspend, exlambda, call/cc}

where call/cc is an estension of call-with-current-cant inuat ion to denote PaiLisp continuation. PaiLisp continuation played an essential role in describing PaiLisp constructs by use of PaiLisp-Kernel.

A PaiLisp continuation is defined for each process; that is, a. PaiLisp continuation of a. process packages up its process-id and continuation. In another words, a. PaiLisp continuation records

the content of the control stack that executes call/cc
and the identity of the process that captured the continuation.

In PaiLisp and PaiLisp-Kernel a PaiLisp continuation can be captured using call/cc as in Scheme and Multilisp. When a. PaiLisp con tinuation is invoked by the same process that. captured it. the resulting behavior is same as in Scheme and Multilisp.

When a PaiLisp continuation is invoked by a different process, the process to execute the rest of computation denoted by the continuation is the process that captured the continuation, discarding its current continuation, and the process that invoked the continuation continues its execution without any disturbance.

With this definition of PaiLisp continuation a PaiLisp program that uses PaiLisp call/cc but no other concurrency constructs produces the same results as in a sequential Scheme. But when a PaiLisp continuation is invoked by a process different from one that captured it, it can be used to give an effect to other processes. Actually it can be used to kill a process to be explained below, and this ability to kill other processes was especially important to define parallel-OR, parallel-AND and other constructs of PaiLisp by use of PaiLisp-Kernel.

2.4 Use of PaiLisp continuation to give an effect to other processes

With the PaiLisp continuation it is possible to give an effect to other processes. Actually we can write a PaiLisp-Kernel program to kill a process. After giving a brief description of PaiLisp-Kernel we explain how to initiate, suspend, resume and kill processes; then we give an example to use PaiLisp continuation to kill processes. More programming examples of PaiLisp and PaiLisp-Kernel will be given in the nest section.

[PaiLisp-Kernel]

PaiLisp-Kernel is defined as Scheme with four concurrency constructs {spawn, suspend, exlambda, call/cc}, where call/cc is PaiLisp's call-with-current-continuation, explained in the preceding paragraph 2.3. Assuming the reader's familiarity on Scheme [Rees86, SprF89] we explain only four concurrency constructs here.

(spawn e): (spawn e) creates a process to compute e, and the parent process captured this statement will be executed concurrently with this newly-created process.

(suspend) : When (suspend) is encountered in the course of esecution of a process its esecution will be suspended, and the execution will be resumed when the continuation creaked by the process is invoked.

(exlambda (x_1 . . . x_n) e_1 . . . e_m) : This statement creates a new queue and an exclusive closure. When the exclusive closure created by this statement is used by a. process, another process that invokes this closure will be suspended in the queue until this closure is released.

(call/cc e): e must be a. procedure of one argument. (call/cc e) creates a. procedure of one argument to denote its current continuation with its process-id. and e will be applied to this procedure. When the continuation is invoked the process that captured it behaves as in the way explained in 2.3.

In order to describe a. concurrent process we usually must know how to 'initiate', 'suspend', 'resu me' a nd 'kill' processes. The initiation will be realized by (spawn

e) to create a process. to compute e and initiate its execution. The temporary suspension of a process will be realized by use of (suspend), and a suspended process will be resumed by invoking a continuation. In order to resume a process suspended by (suspend) its continuation must be created by call/cc and passed to other processes before the suspension. Killing of a process created by (spawn e) will be done by invoking the continuation of e. When the continuation of e is invoked, the process falls into the killed state. The following PaiLisp-Kernel program is an esample to kill and resume processes.

```
(let ((kill (call/cc (lambda (resume)
                       (spawn (call/cc (lambda (k)
                                         (resume k)
                                         ...)))
                       (suspend)))))
  AAA)
```

In this program the variable resume will be bound to the continuation that the body of the let construct is executed after assigning a value into kill. The parent process that executes spawn will be temporarily suspended by (suspend), and the created child-process captures its continuation. Then the value of k will be the continuation which forces the process to terminate. The invocation of resume resumes the execution of the parent process. The variable kill can be used to kill a child-process in execution of AAA.

This ability to kill processes by PaiLisp continuation played an essentially important role to define parallel-OR, parallel-AND and other PaiLisp constructs by use of PaiLisp-Kernel [ItoM90].

2.5 call/ep —A construct for single-use continuation

Concurrent interactions between continuations and concurrency constructs yield some troublesome semantic problems, and it is useful to distinguish between single-use and multiple-use continuations as is discussed in [Hals90].

PaiLisp provides a. new construct (call/ep *e*) which is the PaiLisp call/cc restricted to the single-use continuation. e must be a procedure of one argument, called a receiver, which receives an escape procedure as its argument.

Firstly, we explain call/ep in a. sequential Scheme setting.

Following Halstead [Hals90] we can give an operational explanation of call/cc in the following way. A continuation may be viewed as the stack and register contents that espress the current state and the rest of computation. call/cc captures those stack and register contents, and it copies them into a. continuation object. Whenever the continuation object is applied to a. value *v*, the saved stack and register contents are re-installed as the current, stack and register contents, and v is installed in the appropriate result register, continuing the computation from that state. Then, a return from the original invocation of call/cc occurs, with *v* being the value returned. In this way each invocation leads to a new return from the invocation of call/cc that captured the continuation. In the standard Scheme call/cc the multiple-use continuations are allowed and they leads to several returns from a procedure call. However, call/ep allows only a si ngle-use of continuation in invocation and return.

In case of **call/ep** there is no need to copy the control stack information, for safe-keeping of the working stack. This **call/ep** was installed into a sequential Scheme by 0. Hishida and it has been exhibited that the execution costs of **call/ep** can be reduced about 10%, compared to those of call/cc.

This **call/ep** has been imported into PaiLisp as a restricted version of PaiLisp call/cc, keeping its single-use style feature of continuation.

Usually (call/cc *e*) is used in the following form:

(call/cc (lambda (k) *e*(. . .**k**. . .)))

which means that the continuation captured by call/cc is bound to the variable k, and it may be accessed possibly many times from the inside and outside of *e*(. . .k.. .).

Likewise **(call/ep *e*)** can be used in the following form:

(call/ep (lambda (k) e(. . .k.. .)))

which means that the continuation captured by **call/ep** is bound to the variable k, and it may be accessed at most once within e(. . . k. . .) .

According to our experiences to describe PaiLisp using PaiLisp-Kernel and Scheme programming most cases of using call/cc could be replaced by **call/ep**.

[Behaviors of call/cc and **call/ep**]

In order to explain more details of PaiLisp call/cc and **call/ep** we give several esamples of using call/cc and **call/ep**.

(Example 1) [Continuations used in the downward and single-use style]

Consider the following PaiLisp program [Hals90]:

```
(call/cc
  (lambda (k)
    (+ (future 1)
       (future (k (* 2 3))))))
```

where future is the concurrency construct introduced in Multilisp by Halstead [Hals84], and it is imported into PaiLisp [ItoM90].

In this program the continuations are used in the downward and single-use style. In this case call/cc can be replaced by **call/ep**.

N.B. The "downward" use of continuation means that the continuation is passed deeper into the computation to be used. The "upward" use of continuation means that the continuation is passed out of the context in which it was captured and invoked from elsewhere. This terminology comes from [Hals90].

The future construct has the following meanings: (future *e*) immediately returns a. future-value for e and creates a task to evaluate *e*. The use of the future-value and the evaluation of e may be concurren tly executed. When the evaluation of *e* produces an actual value that value replaces the future-value.

(Example 2) The following receiver-tester program [SprF89] uses the continuations in the upward and multiple-use style.

```
(define receiver
  (lambda (cont inuat ion)
    (continuation continuation)))

(define tester
  (lambda (continuation)
    (print "beginning")
    (call/cc continuation)
    (print "middle")
    (call/cc continuation)
    (print "end") ) )
```

1. Consider (tester (call/cc receiver)). In this case the continuation received by receiver will be invoked at execution of the first (call/cc continuation) of tester, and the continuation of the first (call/cc continuation) will be sent to tester. This continuation can be invoked once again at execution of the first (call/cc continuation). At this point the variable cont inuation is bound to the continuation received by receiver.

   Then the second (call/cc continuation) of tester will be executed, and the continuation of the second (call/cc continuation) will be sent to tester. This continuation will be invoked again at esecutiou of first (call/cc continuation). Thus, the result of (tester (call/cc receiver)) is

   ```
   beginning
   beginning
   middle
   beginning
   end
   ```

2. Consider (tester (call/ep receiver)). In this case, the escape procedure sent to receiver will be invoked within receiver, and after then no invocation will become in effect, so that (tester (call/ep receiver)) produces

   ```
   beginning
   middle
   end
   ```

3. Next, we change tester as follows:

   ```
   (define tester1
     (lambda (continuation)
       (print "beginning")
       (call/ep continuation)
       (print "middle")
       (call/cc continuation)
       (print "end") ))
   ```

   ```
   (define tester2
     (lambda (cont inuat ion)
       (print "beginning")
       (call/cc continuation)
       (print "middle")
       (call/ep continuation)
       (print "end") ) )
   ```

   In case of tester1 the first call/cc of tester is changed to call/ep. and in case of tester2 the second call/cc of tester is changed to call/ep.

Consider (tester1 (call/cc receiver)). After the escape procedure has been sent to tester 1 no invocation to the continuation will take place. The result of (tester1 (call/cc receiver)) is

```
beginning
beginning
middle
end
```

Consider (tester2 (call/cc receiver)). In this case, after call/ep is executed and the escape procedure is sent to tester2 no invocation to the continuation will take place. The result of (tester2 (call/cc receiver)) is

```
beginning
beginning
middle
beginning
middle
end
```

3 Implementation of PaiLisp Continuation and its Evaluations

An interpreter of PaiLisp has been implemented on Alliant FX/80 with eight processor units. The PaiLisp interpreter has been tested and evaluated using the benchmark programs of [MaPTW90], and it has been used in implementation of an algebraic Petri net manipulation system by S. Kawamoto at our group. In this section we esplain how PaiLisp continuations are implemented in this PaiLisp interpreter, and then we give several esamples to show how they work. (More details of PaiLisp interpreter and its applications will be discussed and published elsewhere.)

3.1 Outline of PaiLisp interpreter

The PaiLisp interpreter is implemented on the CONCENTRIX-OS of Alliant FX/80, and it has been programmed in C. The PaiLisp interpreter uses the following registers and the control stacks:

1. general registers: exp, val, fun, argl, unev
2. environment register: env
3. continuation register: cont

The continua&ions are stored in the control stack.

The interpreter reads an expression! evaluates it and prints its result repeatedly. In order to carry out these interpretive processes the following three routines are implemented:

reader: read an expression into val, then jump to a. label denoted by cont

eval_dispatch: evaluate an expression in exp under an environment env, and set. its value into val, then jump to a. label denoted by cont

printer: print the espression in val, then jump to a label denoted by **cont**

The C program of **eval_dispatch** is as follows:

```
int eval_dispatch(void){
    if self_evaluatingp(exp)
        go(ev_self_eval);
    if variablep(exp)
        go (ev,variable) ;
    if (length(exp) <= 0)
        go (unknown,exp) ;
    fun = car(exp);
    if syntaxp(fun)
        go(sym_syntax(fun));
    if macrop (fun)
        go (expand-macro) ;
    if (cdr(exp) == Nil)
        go (ev,no,args) ;
    go(ev_application);
}
```

3.2 PaiLisp process and continuation

In PaiLisp interpreter a process is defined as an object with

its process name
its current value
its current state
its current continuation
the information on the exclusive resources used by the process

A PaiLisp process is actually realized as a process object with the following structure:

current continuation			
list of continuations to be resumed at termination of esecution			
the final value of the process			
pointers to other processes			
type	state	lock- byte	

A PaiLisp continuation is realized as an **object** with the following **structure:**

pointer to the process which the continuation belongs to
pointer to the control information that denotes the content of the stack at execution of continuation
return value

3.3 On executions of PaiLisp processes

A process esecution means an **esecution** of its current. continuation. In an idea.1 case that an infinite number of processors are available **the** state transitions of PaiLisp processes will **become as** in Figure 1. **However, in** an actual **case** that only a. finite number of processors are available the state transitions will become as in Figure 2.

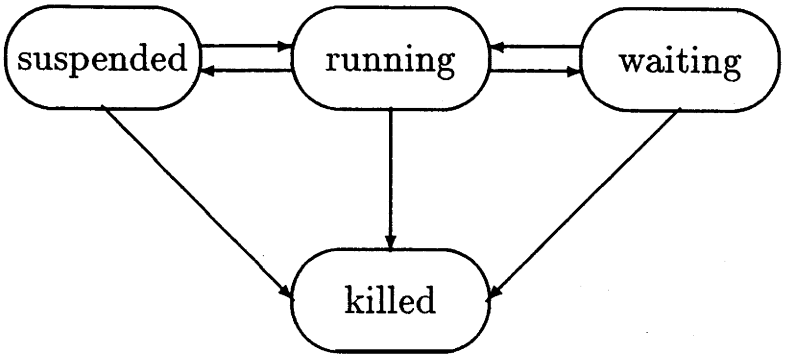


Figure 1: The state transitions in an ideal case.

That is, an executable continuation resides in the queue, and an available non-busy processor executes a continuation in the queue. When a continuation is executed the control information of its continuation will be moved into the stack area and the return value will be stored in the val register. The processor allocation of a resumed continuation has higher priority than that of a newly-created continuation. The current continuation of a process may be altered by invoking continuations of other processes. An invoked continuation will be esecuted only when a processor is allocated to it. An invoked continuation will be eventually esecuted unless it is hampered by other processes.

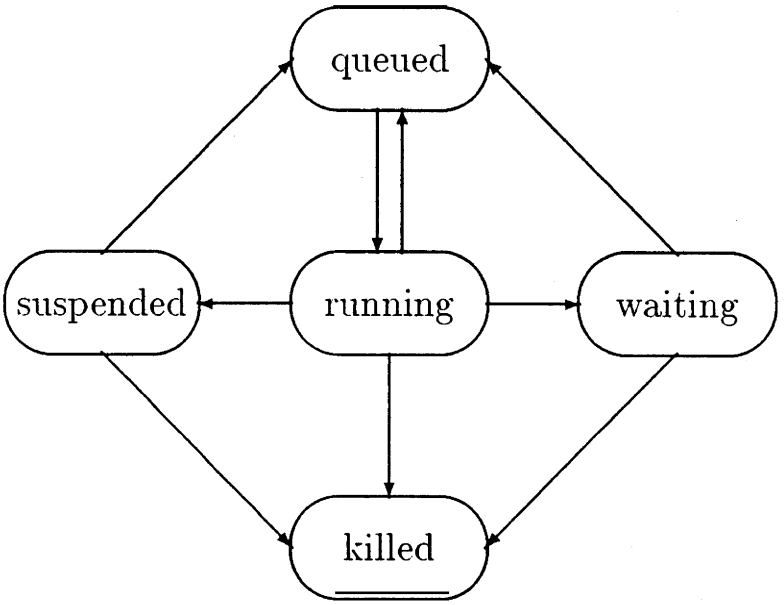


Figure 2: The state transitions in an actual case.

3.4 Realizations of spawn, suspend, and call/cc

The basic concurrency constructs of PaiLisp-Kernel(spawn, suspend, `exlambda`, call/cc) are realized as the C programs. The C programs of spawn, suspend

and call/cc are given in Figure 3, but the C program of **exlambda** is omitted here, since it is rather complicated and lengthy.

[spawn]

```
int ev_spawn(void){
    cons(exp, Term, cdr(exp));
    allocp(val, Nil, Nil, Nil, PROCESS);
    create_cont(argl, exp, env, val);
    add-new-cont(arg1);
    return-value;
}

int terminate_process(void){
    lock(p_lock(process));
    killed-state(process);
    unlock(p_lock(process));
    go(take_cont);
}
```

[suspend]

```
int suspend(void){
    go(take_cont);
}
```

[call/cc]

```
int cwcc_apply(void){
    fun = car(argl);
    stack_to_list(argl);
    allocate-cont(exp, process, argl, Nil);
    cons(argl, exp, Nil);
    go(apply_dispatch);
}

int invoke_cont(void){
    tmp = k_p(fun);
    val = car(argl);
    lock(p_lock(tmp));
    allocate-cont(fun, tmp, k_stk(fun), val);
    p_cc(tmp) = fun;
    if (p_stat(tmp) & RUNNING)
        run_cont(fun);
    unlock(p_lock(tmp));
    popcont; .
    return-value;
}
```

Figure 3: The C programs of spawn, suspend, and call/cc.

3.5 Evaluations of PaiLisp continuation

With the above implementations of PaiLisp and PaiLisp continuation we consider the overheads of PaiLisp continuation. In **execution** of **(call/cc** e**) the result** of

evaluating the argument e must be a function of one argument and a new continuation will be created. The copy of the content of the control stack will be set into the newly-created continuation, together with the process name that captured call/cc. The list of the resultant continuations will be sent to the function e. On invocation of continuation invoke,cant that invokes the continuation will be executed, receiving its argument, and invoke-cant returns the value of its argument. If the process to which the continuation belongs is not in the killed state the current continuation will be changed to the newly-invoked continuation, discarding the previous continuation. Thus the pointer of the current continuation must be the pointer to the newly-created continuation object. The invocation of a continuation will be checked by a processor which execute a continuation of a process, whenever the eval routine **eval_dispatch** returns a value. If the process state indicates several occurrences of invocations, the processor will be allocated to another executable continuation. These behaviors of capturing and invoking PaiLisp continuations may be illustrated as in Figure 4.

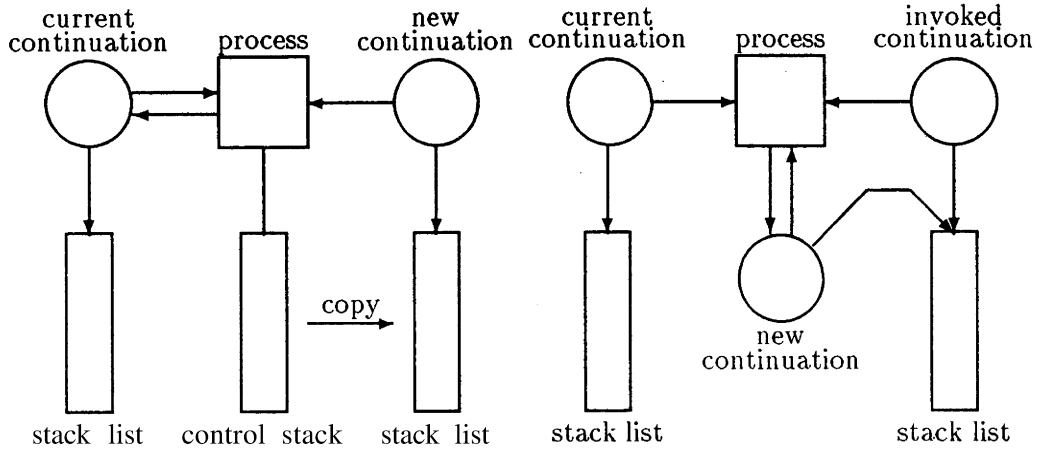


Figure 4: Behaviors of capturing and invoking PaiLisp continuation.

From these behaviors of PaiLisp call/cc we can see

1. the cost of capturing a PaiLisp continuation is same as Scheme continuation.
2. when a PaiLisp continuation is invoked by the same process that captured it, the cost of invoking the PaiLisp continuation is essentially same as in Multilisp.
3. when a PaiLisp continuation is invoked by a different. process, the cost of invoking the PaiLisp continuation may be high because of
   - checking process states
   - switching continuations
   - allocating processors to processes

   However the actual cost, of PaiLisp continuations is not high according to our experiences to be explained below.

Let us consider the following simple program:

```
(let ((c <number-of-iterations>))
  (loop (cond ((zero? c) (exit))
              (else (set! c (- c 1))))))
```

where <number-of -iterations> means the number of looping.

Using call/cc we can write the following program to perform an essentially same looping computation.

```
(let ((c <number-of-iterations>) (tag 'init))
  (call/cc (lambda (k) (set! tag k)))
  (cond ((zero? c) (exit))
        (else (set! c (- c 1)) (tag 'dummy))))
```

Changing <number-of-iterations> we obtained the following experimental results of executing the above programs on the PaiLisp interpreter.

<number-of-iterations>	[diff-process]	[same-process]	[loop]
1000	1.13 sec	1.03 sec	0.76 sec
10000	11.22	10.28	7.62
100000	112.19	102.82	77.07

where [different-process] means that the continuation was invoked by a different process, [same-process] means that the continuation was invoked by the same process that captured it, and [loop] means the above program of using the loop construct. The results of [same-process] are essentially same with those of using Scheme continuations. This shows that the execution overheads of PaiLisp continuations are about 10%, compared to Scheme continuations.

3.6 Running PaiLisp continuations on PaiLisp interpreter

In this section we give several PaiLisp programs of using PaiLisp call/cc. All these programs were actually executed on PaiLisp interpreter.

(Example 3) [downward and single-use style continuation]
This is the example mentioned in [Hals90].

```
(call/cc (lambda (k) (+ 1 (k (+ 2 3)))))
→ 5
```

Next we insert future into this program.

```
(call/cc (lambda (k) (future (+ 1 (k (+ 2 3))))))
→ [future-value]
→ 5

(call/cc (lambda (k) (+ 1 (future (k (+ 2 3))))))
→ 5
```

These three programs produce the same result according to the PaiLisp interpreter.

(Example 4) [upward and single-use style continuation]

```
(define a
  (let ((s '()) (r '()))
    (lambda ()
      (set! r (call/cc (lambda (k) (set! s k) #f)))
      (print "This is a. ")
      (cond (r (r #f))
            (else (b s))))))

(define b
  (let ((s '()) (r '()))
    (lambda (c)
      (set! r (call/cc (lambda (k) (set! s k) #t)))
      (print "This is b.")
      (cond (r (c s))
            (else (print "That's all."))))))

(a)
This is a.
This is b.
This is a.
This is b.
That's all.
```

(Example 5) [upward and multiple-use style continuation]

```
(let ((c 0) (tag '()))
  (set! c (call/cc (lambda (k) (set! tag k) 3)))
  (print c)
  (if (zero? c) 0 (tag (- c 1))))

3
2
1
0
```

Inserting future in front of call/cc we get the following program.

```
(let ((c 0) (tag '()))
  (set! c (future (call/cc (lambda (k) (set! tag k) 3))))
  (print c)
  (if (zero? c) 0 (tag (- c 1))))

3
```

At execution of (tag (- c 1)) the future-value is determined already and the process that created continuation is terminated, so that the invocation of its continuation will give no effect.

(Example 6) [parallel-OR]

Originally the PaiLisp continuation was invented to describe the PaiLisp constructs like-parallel-OR, parallel-AND, etc. by use of PaiLisp-Kernel. The parallel-OR is defined in PaiLisp as follows:

syntax: (par-or e_1 e_2 . . . e_n)

semantics: The expressions e_1 , e_2, . . . , e_n will be executed in parallel.

> Whenever one of e_1 , e_2, . . . , e_n gives non-NIL par-or returns this non-NIL value as its result, and the executions of other expressions must be killed for termination. If none of e_1, e_2, . . . , e_n becomes non-NIL, then par-or returns false as its value.

A complete PaiLisp-Kernel program to describe par-or is given in Appendix. In this description the actions of killing processes are realized by use of PaiLisp continuation. However in the PaiLisp interpreter par-or is directly realized as a C program. It is possible to compare par-or of PaiLisp interpreter and the PaiLisp-Kernel program of par-or, since both of them are executable on PaiLisp interpreter.

Table 1 gives the result of running (OR (fib e_1) (fib e_2) (fib e_3))
using 1) sequential or
2) par-or of PaiLisp
and 3) k-par-or which is the PaiLisp-Kernel program in Appendix.
where (fib e) is the following Scheme program to compute a Fibonacci number of e.

```
(define fib
  (lambda (n)
    (if (< n 2)
        1
        (+ (fib (- n 1)) (fib (- n 2))))))
```

	(e_1, e_2, e_3)		
OR	(10, 15, 20)	(15, 10, 20)	(20, 10, 15)
or	0.1s sec	*2.05* sec	22.5 sec
par-or	0.19	0.1s	0.19
k-par-or	0.21	0.21	0.21

Table 1: The result of running (OR (fib e_1) (fib e_2) (fib e_3))

From this result we can see that the overhead of k-par-or is about 0.02 sec, compared to par-or: and the sequential or depends on the order of arguments of OR-operation.

4 Concluding Remarks

[1] The PaiLisp continuation was introduced as an extension of Scheme continuation into concurrency. PaiLisp continuation can be created by use of call/cc in PaiLisp, and its restricted version `call/ep` serves as a. construct of a. single-use style continuation. They have been implemented in PaiLisp interpreter, but it remains to find good and efficient implementation strategies of PaiLisp continuations. Also there are some possibilities to extend PaiLisp continuation further. For example we may be able to imagine an extended call/cc which allows multiple capturings and invokings of continuations in multi-threads of parallel computation.

[2] PaiLisp continuation may be called a parallel continuation, in short, P-continuation. In this paper we explained the operational meanings of P-continuation informally. There are at least three directions to give formal semantics of P-continuation:

1. Plotkin's structural operational semantics of P-continuation
   We gave a structural operational semantics for a subset of PaiLisp-Kernel, called mini-PaiLisp-Kernel, with the following syntactic definition:

   E ::= **K|I|(E E*)|(lambda** (I*) E **E*)|(lambda (I*.I)** E **E*)**
   | (if E_0 E_1 E_2) | (if E_0 E_1) | (set! I **E)**
   |(spawn E)|(suspend)|(exlambda (I*) E **E*)|(call/cc E)**

   where K is the set of constants and I is the set of identifiers. However our SOS semantics for the above language became complicated to treat P-continuation although a complete SOS semantics of mini-PaiLisp-Kernel is reported in M.S. thesis of M. Umemura(March, 1990). A clean SOS style formal semantics of P-continuation and mini-PaiLisp-Kernel is open for future study.

2. Process calculus description of P-continuation
   There have been proposed a number of powerful process calculi like π-calculus of Milner-Parrow-Walker, CHOCS of Thomsen and y-calculus of Berry-Boudol. According to our preliminary experiences it seems that all of them are rather weak to describe semantics of P-continuation and mini-PaiLisp-Kernel. The first author is currently working to use a-calculus to give a semantics of P-continuation. The a-calculus introduced by the first author is the sum of CHOCS and an extended y-calculus with n-ary cooperation operators. In another words the a-calculus is a process calculus with the associative parallel composition and the non-associative parallel compositions to treat processes as the first class objects of the calculus.

3. Logical approach for formal semantics of P-continuation
   There are several interesting logical approaches to give formal semantics of sequential continuations by T. Griffin, C. Murthy, and S. Nishizaki. Is there any powerful logical framework to give a formal semantics of P-continuatiou in a logical setting?

[3] Halstead [Hals90] proposes the criteria for "continuation". Halstead's criteria are designed to check fitness for Multilisp. His criteria, are as follows:

1. Programs that use call/cc but no concurrency constructs should yield the same results as in a sequential Scheme.

2. Programs that use continuations exclusively in the single-use style should yield the same results as in a sequential Scheme.

3. Programs should yield the same results as in a sequential Scheme. even if future is wrapped around arbitrary subexpressions, with no rest ric tions on how con ti n uations are used.

He states that Katz-Weise approach to change the definition of future is the best for Multilisp [Hals90].

The following questions on P-continuation arise:

- Can we think of any good criteria for P-continuation?
- Should we change the definitions of concurrency constructs for semantic safekeeping with P-continuation?
- Should we change the meanings of P-continuation to meet semantics of concurrency constructs?

Similar questions may be raised to PaiLisp `call/ep`. In case of `call/ep` we may have the following additional question.

- Can we give an algorithm to detect which call/cc can be replaced to `call/ep`?

References

[Fell88] M. Felleisen, The theory and practice of first-class prompts, 15th Annual ACM Symp. on Principles of Programming Languages, San Diego, Ca., pp. 180–190(Jan. 1988)

[FeWFD88] M. Felleisen, M. Wand, D. Friedman, B. Duba, Abstract continuations: A mathematical semantics for handling full functional jumps, ACM Conference on Lisp and Functional Programming, pp. 52–62 (1988)

[Hals84] R. H. Halstead, Jr., Implementation of Multilisp: Lisp on a multiprocessor, Conf. Record of 1984 ACM Symposium on Lisp and Functional Programming, pp. 9–17(1984)

[Hals90] R. H. Halstead, Jr., New ideas in parallel Lisp: Language, implementation, and programming tools, Springer LNCS, 441, pp. 2–57(1990)

[HieD90] R. Hieh, R. Ii. Dybvig, Continuations and concurrency, 1990 ACM Conf. Principles and Practice of Parallel Programming, pp. 128–136(1990)

[HisI88] 0. Hishida., T. Ito, On importing non-local esit, structure and package into Scheme, Proceedings of 5th Conference of Japan Society for Software Science and Technology, pp. 325–328(September 1988) (in Japanese)

[Ito91] T. Ito, Lisp and parallelism, in "Artificial Intelligence and Mat hematical Theory of Computation", pp. 187–206, Academic Press(1991)

[ItoM90] T. Ito, M. Matsui, A parallel Lisp language PaiLisp and its kernel specification: Springer LNC'S, 441, pp. 58–100(1990)

[MaPTW90] I. A. Mason, J. D. Pehoushek, C. Talcott, J. Weening, Programming in QLisp, Stanford University, STAN-CS-90-1340(1990)

[Rees86] J. Rees, W. Clinger(Eds.), The revised[3] report on the algorithmic language Scheme, ACM SIGPLAN Notices, 21, pp. 37–79(1986)

[Sch86] D. Schmidt, Denotational Semantics: A Methodology for Language Development, Allyn and Bacon, Newton, Mass. (1986)

[SprF89] G. Springer and D. Friedman, Scheme and the Art of Programming, The MIT Press, pp. 516–578(1989)

[Stoy81] J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Languages, The MIT Press, Cambridge, Mass.(1981)

This work was partially supported by Grant-in-Aid for Scientific Research (A)01420029 and (A02)02249102, under The Ministry of Education, Science and Culture, Japan.

Appendix: PaiLisp-Kernel program of par-or

```
(macro k-par-or
  (lambda (form)
    '(call/cc (lambda (*return*)
       (let* ((*process-list*'())
              (*kill-lock* (exlambda (e) (e)))
              (*l* ,(length (cdr form)))
              (*l-unlock*
                (exlambda ()
                  (if (= *l* 1)
                      (*return* #f)
                      (set! *l* (- *l* 1))))))
         (letrec ((*kill-all*
                    (lambda (k pl)
                      (cond ((null? pl) 'dummy)
                            ((eq? k (car pl)) (*kill-all* k (cdr pl)))
                            (else ((car pl) 'dummy)
                                  (*kill-all* k (cdr pl)))))))
           (let ((*start-process*
                  (lambda (e)
                    (*kill-lock*
                     (lambda ()
                       (call/cc (lambda (r)
                          (spawn
                            (call/cc (lambda (k)
                              (set! *process-list* (cons k *process-list*))
                              (r 'dummy)
                              (let ((res (e)))
                                (cond (res (*kill-lock*
                                             (lambda ()
                                               (*return* res)
                                               (*kill-all* k *process-list*))))
                                      (else (*l-unlock*))))))))))))
             (suspend))))))
           ,@(map (lambda (x) '(*start-process* (lambda () ,x)))
                  (cdr form))
           (suspend)))))))
```

Graph Reduction and Lazy Continuation-Passing Style

Chris Okasaki Peter Lee David Tarditi

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

1. Introduction

Some implementations of lazy functional programming languages compile programs into graphs [25]. The use of graphs makes it easy to express normal-order evaluation and the sharing of bindings, thus providing a simple and relatively efficient method for implementing lazy evaluation. Augusteijn and van der Hoeven [2] and Koopman and Lee [16] have shown how self-modifying code can model directly and efficiently the self-reducing nature of such graphs. In these approaches, graphs are compiled into threaded code in which reductions of the graph are accomplished by modifying the code stream.

In a different arena of programming language implementation, interest has steadily increased during the past fifteen years in the use of continuation-passing style (CPS) as a compiler intermediate representation [24, 17, 1]. Techniques for converting both applicative-order X,-terms and normal-order λ_n-terms are well known [23, 22, 7], but to the best of our knowledge CPS-conversion has never previously been described for *lazy* X-terms. In this paper we present a lazy CPS transformation and examine its relationship to graph reduction.

We begin by reviewing the basic concepts of graph reduction. We then present in more detail self-modifying graph reduction and show how graphs resemble continuations. Looking at this resemblance from the other direction, we modify Plotkin's CPS transformation for the normal-order λ-calculus to account for laziness, and then show how the result resembles graph reduction, especially the self-modifying approaches. Finally, we describe where graph reduction and lazy CPS fit into the broader landscape of implementation techniques for lazy evaluation.

2. A Brief Review of Graph Reduction

Turner proposed the use of graph reduction to implement lazy functional programming languages [25]. He described what is referred to as combinator-graph reduction, which is based on the well-known fact that any closed X-expression can be transformed into an expression involving only applications and combinators.[1]

Such a combinator expression is easily represented as a binary tree, where the interior nodes represent the applications and the leaves represent the combinators. Furthermore, by moving to a binary directed graph, the

This research was partially supported by the National Science Foundation under PYI grant #CCR-9057567, with matching funds from Bell Northern Research. David Tarditi is supported by an AT&T PhD Scholarship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the US Government, or AT&T.

[1] A combinator is simply (the name of) a closed λ-expression [4]. In addition to being closed, there is typically the stipulation that &abstractions not occur in the argument position of an application.

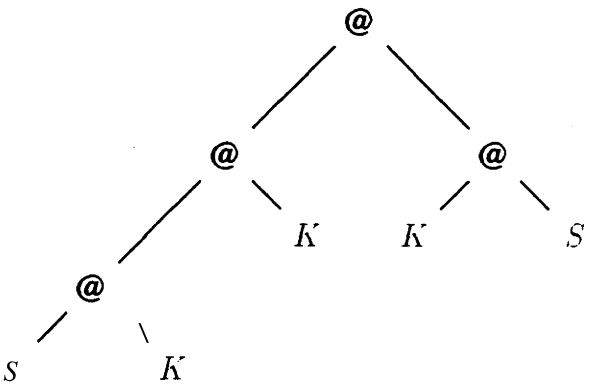


Figure 1: Representation of the graph for ((S K **K)** (K S)).

occurrence of common **subexpressions** can be represented by **subgraph** sharing and recursion by cycles. The (possibly cyclic) path of left-branches starting from the root node is referred to as the *spine* of the graph.

In combinator graphs, the definitions of the combinators denote graph-rewriting rules, and executing programs becomes a process of graph reduction. The spine of the graph is traversed and the spine nodes are pushed onto the "spine stack." When a combinator is encountered, the graph is rewritten according to the corresponding rewrite rule. This process is repeated on the new graph. Program execution terminates if and when an irreducible graph is produced. The consistent reduction of the spine corresponds to the "leftmost outermost" rule of normal-order reduction. This, in conjunction with the sharing and destructive update of pointers to subgraphs, leads to the so-called "lazy" evaluation of functional programs.

As an example, consider the reduction of the graph shown in Figure 1, corresponding to the combinator expression, ((S K K) (K S)). Here we can see that the graph is implemented as a collection of application nodes (depicted by **"@")** and combinator nodes. Application nodes contain references to other nodes, whereas the combinator nodes contain a token denoting a graph-rewriting **action.**[2]

When a graph is reduced, the spine nodes are pushed onto the spine stack, The spine stack provides references to the arguments required by a combinator, and a combinator consumes part of the spine stack in addition to rewriting a graph node. The result of such a rewrite is depicted in Figure 2. Note in this figure that a node has been destructively updated by this rewriting. This allows the result of the reduction to be shared by several parts of the program, thereby leading to "lazy" behavior.

To make things more concrete, consider the following representation of combinator **graphs:**[3]

```
stack   =  Node list
Answer  =  program answers (unspecified)
Node    =  Comb of Stack — Answer
        |  App of Node × Node
```

Note that we use a list to represent the spine stack and that combinators are represented by "code" that computes a program answer from a *spine* stack. A graph *is* **reduced** by *forcing* it with the empty stack, where the force function is

[2] **We ignore the issue of strict combinators which might be used, for instance, for arithmetic on integer baseconstants. Hence for the purposes of this paper there arc no nodes that contain base-constant values.**

[3] **Throughout this paper, code exampks will be presented in a notation loosely based on Standard ML.**

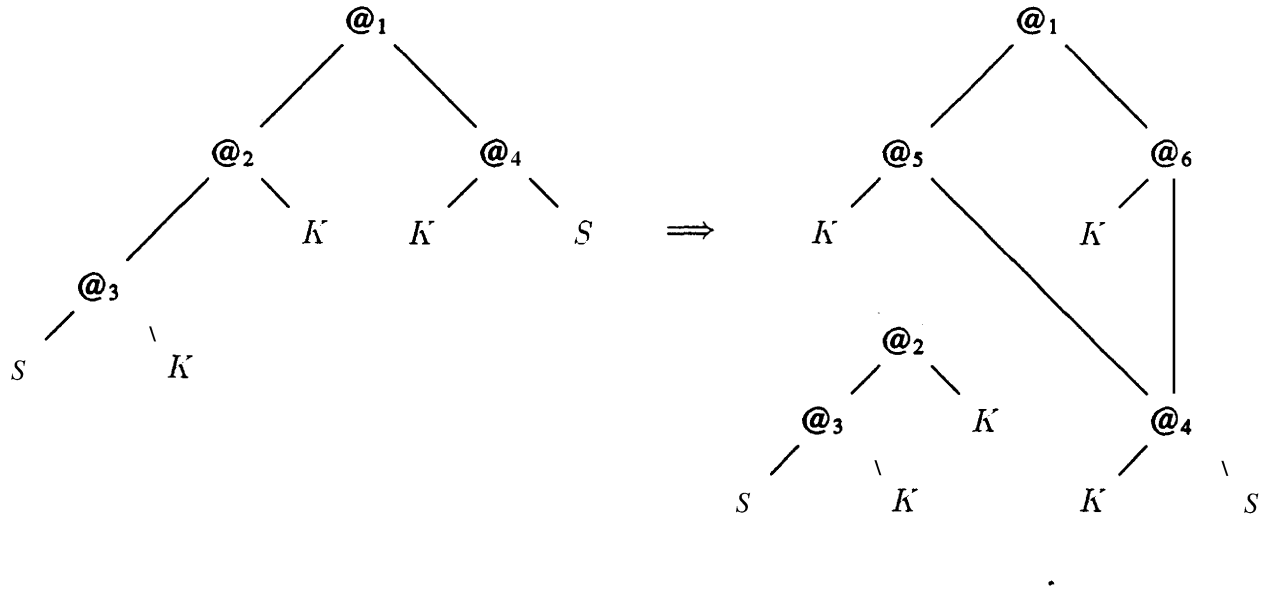


Figure 2: The original and rewritten graphs. **Note** that node 1 has been destructively updated and node 4 is now shared. Nodes 2 and 3 are now garbage (unless referred to by some other **unshown** node). Nodes 5 and 6 were created by this rewrite.

defined by

$$\begin{array}{lll} \textbf{force}\ g\ s & = & \text{casegof Combc} \Rightarrow \text{cs} \\ & & \quad \textbf{App}(m,n) \Rightarrow \textbf{force}\ m\ (n :: \text{s}) \end{array}$$

The notation, *"n :: s"* denotes the cons of *n* and s, that is, the result of pushing the argument node onto the spine stack.

The above **definitions** are incomplete in that they fail to account for the need to share and update nodes destructively during graph rewrites. For this, we borrow Standard **ML's [19]** notation for references to provide a mechanism for destructive updates[4] and modify the representation as follows:

$$\begin{array}{lll} \text{stack} & = & \textbf{(Node ref) list} \\ \text{Node} & = & \text{Comb of Stack} \rightarrow \text{Answer} \\ & | & \textbf{App of Node ref} \times \text{Noderef} \end{array}$$

Then the force function is given by

$$\begin{array}{lll} \text{force g s} & = & \textbf{case !}g\ \textbf{of Combc} \Rightarrow \text{cs} \\ & & \quad \text{App}(m,n) \Rightarrow \text{force}\ m\ (\text{g} :: \text{s}) \end{array}$$

The reason g is pushed onto the stack instead of *n* is so that it can serve as the target for an update. The value of *n*

---

[4]Standard ML uses the following notation for references:

- "τ ref" denotes the type of references to objects of type τ,
- "ref v" denotes the creation of a ref cell with value v,
- "! n" denotes the extraction of a value from a ref cell (a "dereference" operation),
- "$n := v$" denotes the assignment of v to the ref cell n, and
- ";" denotes sequencing.

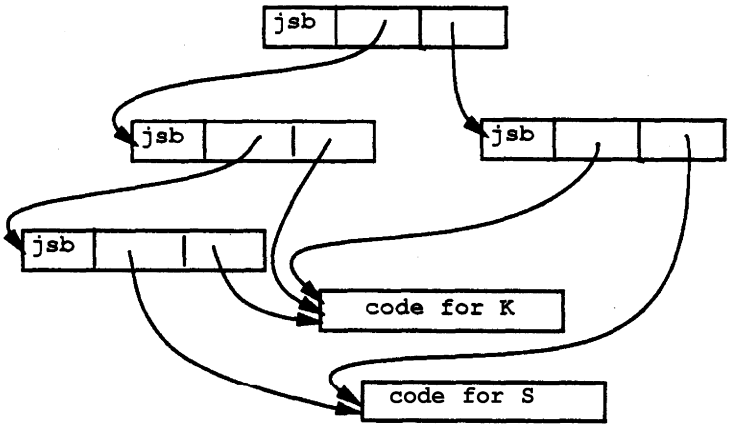


Figure 3: A graph as self-modifying code.

may be extracted from g when needed by an operation referred to as fetch, where

$$\textbf{fetch}\ (\textbf{ref}\ (\textbf{App}(\_, n))) \quad = \quad n$$

Now the S combinator (which corresponds to the graph rewrite rule S $f\ g\ x \Rightarrow (f\ x)\ (g\ x))$ *can be* defined as follows:

$$\begin{aligned} S \quad = \quad & \textbf{Comb}(\lambda s.\ \text{case s of } n_1 :: n_2 :: n_3 :: s' \Rightarrow \\ & \quad n_3 := \textbf{App}(\textbf{ref App}(\textbf{fetch } n_1, \textbf{fetch } n_3), \textbf{ref App}(\textbf{fetch } n_2, \textbf{fetch } n_3)); \\ & \quad \text{force } n_3\ s') \end{aligned}$$

This code assumes that there will always be enough arguments on the stack. If not, the graph is irreducible and program execution terminates.

3. Graphs as Self-Modifying Code

A number of techniques for compiling programs into graphs [3, 12] and efficiently reducing them [8, 14] have been proposed. Of particular interest are those approaches that use self-modifying code to model the self-reducing nature of combinator graphs [2, 16]. In these approaches, the key observation made is that *the spine stack is* ***actually*** *the subroutine return stack* for a threaded program. Thus, the left-hand-side field of each application node can be implemented by a subroutine call. As these subroutine calls are executed, the "return addresses" that are saved on the subroutine-return stack are references to the right-hand-side fields of the spine-which is exactly the desired behavior. Combinators can be represented as pointers to code sequences that perform graph rewrites by consuming return-stack elements and modifying the code stream in an appropriate manner. Figure 3 shows the representation of our example graph in this scheme. Note that j `sb` is the VAX mnemonic for the "jump to subroutine" instruction, which stacks the value of the address following the `j sb` instruction and then transfers control.

Architectures that allow self-modifying code and provide a subroutine-call instruction, such as older VAX computers, allow a direct implementation of this approach. Great efficiency is gained from the use of the native hardware's instruction-decoding mechanism to implement the case analysis on graph nodes. Unfortunately, in practice self-modifying code is seldom feasible, especially on modem architectures that enforce a separation of cache memories for instructions and data [16]. Furthermore, many RISC architectures do not provide a subroutine-call instruction, and may further complicate matters by introducing branch delays.

4. Graphs as **Continuations**

It is natural to think of self-reducing graphs as self-modifying programs. Doing so gives rise to a notion of "direct execution": graphs are no longer interpreted as described in Section 2, but rather executed directly as shown in Section 3. The benefit in execution time comes from the fact that explicit case analysis on graph nodes is no longer necessary.

A similar effect is also obtained by viewing graphs as continuations. In this view, a graph (and hence each graph node) is a continuation which takes a stack and performs the remainder of the computation. This is reflected in the following representation:

$$\text{Node} = (\text{Stack} \rightarrow \text{Answer}) \text{ ref}$$

(The **ref** simply allows continuations to be destructively updated.) Again, there is no case analysis but instead of performing a subroutine call to a node, one "throws" (a spine stack) to a continuation. **The** continuation for an application stacks both itself and the argument continuation and throws the result to the left-branch continuation:

$$\textbf{app}\ m\ n = \textbf{fix}\ r.\ \textbf{ref}\ (\lambda s.\ \textbf{force}\ m\ ((r, n) :: s))$$

where

$$\begin{aligned} \text{Stack} &= (\textbf{Node} \times \textbf{Node})\ \textbf{list} \\ \textbf{force}\ g\ s &= g\ s \end{aligned}$$

Both the application continuation itself and the argument continuation must be stacked because in this representation, application nodes *are opaque-we* can no longer "look inside" the application to extract the argument like the fetch operation does in Section 2. **The** fix is required for the continuation to know its own "location."

With this representation, the S combinator is **defined** as follows:

$$\begin{aligned} S = \textbf{ref}\ (\lambda s.\ \textbf{case}\ s\ \textbf{of}\ \ & (\_, n_1) :: (\_, n_2) :: (g, n_3) :: s' \Rightarrow \\ & \textbf{update}\ g\ (\textbf{app}\ n_1\ n_3)\ (\textbf{app}\ n_2\ n_3); \\ & \text{force g s')} \end{aligned}$$

where

$$\text{updategmn} = g := \lambda s.\ \textbf{force}\ m\ ((g, n) :: s)$$

Note that update is just like app except that it overwrites an existing cell instead of allocating a new one.

Looking again at the self-modifying code approach, we can now view the use of subroutine calls as a shorthand, or optimization. The VAX j**sb** instruction has the convenient effect of stacking the current node and then "throwing" to the left branch. (Actually, the address of the right **subnode** is stacked, but simple address arithmetic allows us to compute the address of the node itself **from** this.) There is no need to stack both the current node and the argument node because of course, at the machine level, the latter may be extracted from the former

5. Lazy Continuation-Passing Style

Having noted that graphs are continuations, it is natural to wonder if graph reduction can be written in continuation-passing style. We answer in the **affirmative** but **first** we must describe CPS-conversion for lazy programs.

Plotkin's transformation of (normal-order) λ_n**-terms** into CPS (see **[22])** is given by the following rules:

$$\begin{aligned} [x] &\Rightarrow x \\ [\lambda x.\ M] &\Rightarrow \lambda k.\ k\ (\text{Ax.}\ [M]) \\ [M\ N] &\Rightarrow \lambda k.\ [M]\ (Am.\ m\ [N]\ k) \end{aligned}$$

Before modifying this transformation for laziness, we first examine more closely how Plotkin's transformation actually works by isolating the essential operations in the generated programs and encapsulating those **operations**

within auxiliary functions with (hopefully) more descriptive names. Modifying the transformation then consists of redefining the relevant auxiliaries.

The evaluation of an expression is delayed by packaging the expression into a *thunk.* In this context, a thunk is a function that takes a continuation, evaluates the expression, and throws the result to the continuation. A thunk is *forced* by applying it to a continuation. We make each force explicit by rewriting the rule for application as follows:

$$[M\ N] \Rightarrow \lambda k.\ \text{force}\ [M]\ (\lambda m.\ \text{force}\ (m\ [N])\ \text{k})$$

where

$$\textbf{forcemk} = m\ k$$

Inside the application rule, the thunk in the function position is forced with a continuation that accepts a function value and applies it to the argument thunk. This continuation is given the name **arg.** Inside the lambda-abstraction rule, the continuation is applied to a value. This is given the name return. The transformation may then be rewritten

$$\begin{array}{l} [x] \quad\ \Rightarrow x \\ [\lambda x.\ M] \Rightarrow \textbf{return}\ (\text{X}x.\ [M]) \\ [M\ N] \quad \Rightarrow \lambda k.\ \text{force}\ [M]\ (\text{arg}\ [N]\ k) \\ \qquad\quad \Rightarrow \text{force}\ [M] \circ \text{arg}\ [N] \end{array}$$

where

$$\begin{array}{lcl} \textbf{return}\ v & = & \lambda k.\ \text{k}\ v \\ \text{argnk} & = & \lambda m.\ \textbf{force}\ (m\ n)\ \textbf{k} \end{array}$$

Finally, we give the names **fun** and app to the right-hand-sides of the lambda-abstraction and application rules, respectively, and write the **final** version of Plotkin's transformation as follows:

$$\begin{array}{l} [x] \quad\ \Rightarrow x \\ [\lambda x.\ M] \Rightarrow \textbf{fun}\ (\text{X}x.\ [M]) \\ [M\ N] \quad \Rightarrow \text{app}\ [M][N] \end{array}$$

where

$$\begin{array}{lcl} \textbf{fun}\ f & = & \text{returnf} \\ \textbf{app}\ m\ n & = & \text{force}\ [M] \circ \text{arg}\ [N] \end{array}$$

Now, in order to obtain a lazy transformation, we must arrange that when a thunk is forced for the **first** time, the result is saved so that it may immediately be returned whenever the thunk is subsequently forced This is accomplished by physically replacing the thunk with a new thunk that immediately returns the desired result. But before we can do this, we must first make the thunks mutable. Only the auxiliary **functions** fun, app, and force **need be modified.**

$$\begin{array}{lcl} \textbf{fun}\ f & = & \textbf{ref}\ (\textbf{return}\ f) \\ \textbf{app}\ m\ n & = & \textbf{ref}\ (\textbf{force}\ [M] \circ \textbf{arg}\ [N]) \\ \textbf{force}\ m\ k & = & !\ m\ k \end{array}$$

Next, we must **specify** the actual update. This could be associated with the **force** operation, but we do not wish to update thunks every time they are **forced** nor do we wish to require an explicit test to determine whether the thunk has been previously forced. Instead, we add to each thunk a continuation which captures the result, updates the thunk, and passes on the result to the next continuation. Note that we do not need to save the results of evaluating a lambda-abstraction, since thunks for such expressions are already in the desired form, that is, they immediately return a value. Only the auxiliary function app need be **modified.**

$$\textbf{app}\ m\ n\ = \text{fix}\ r.\ \text{ref}\ (\text{force}\ [M] \circ \text{arg}\ [N] \circ \text{update}\ r)$$

where

$$\textbf{update}\ r\ k\ =\ \lambda v.\ (r\ :=\ \textbf{return}\ v;\ k\ v)$$

$$
\begin{array}{lll}
[x] & \Rightarrow & x \\
[\lambda x.\, M] & \Rightarrow & \mathbf{fun}\,(\lambda x.\ [M]) \\
[M\ N] & \Rightarrow & \mathrm{app}\ [M][N]
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{fun}\, f & = & \mathbf{ref}\,(\mathbf{return}\, f) \\
\mathbf{app}\, m\, n & = & \mathbf{fix}\, r.\, \mathbf{ref}\,(\mathbf{force}\, m \circ \mathit{argn} \circ \mathrm{updater}) \\
\mathbf{force}\, m\, k & = & !m\ \ k \\
\mathbf{return}\, v & = & \mathit{Xk}.\, k\, v \\
\mathbf{arg}\, n\, k & = & \mathrm{Am}.\ \mathrm{force}\,(m\, n)\, k \\
\mathbf{update}\, r\, k & = & \lambda v.\,(r\ :=\ \mathbf{return}\, v;\ \ k\, v)
\end{array}
$$

$$
\begin{array}{lll}
[x] & \Rightarrow & x \\
[Xx.\, M] & \Rightarrow & \mathrm{ref}\ (Xk.\, k\ (\lambda x.[M])) \\
[M\ Iv] & \Rightarrow & \mathbf{fix}\, r.\, \mathbf{ref}\ (![M]\,(\lambda m.\, !(m\,[N])\,(\lambda v.\,(r := \lambda k'.\, k'\, v;\, k\, v))))
\end{array}
$$

Figure 4: The lazy CPS transformation (with auxiliaries and in expanded form)

This completes the modifktions for laziness. The entire transformation is shown in Figure 4.

An important theoretical property of Plotkin's normal-order CPS transformation is independence of evaluation order; it does not matter whether the the resulting programs are evaluated using call-by-value or call-by-name [22]. Unfortunately, this property no longer holds for the lazy CPS transformation. The way that assignments are used to model laziness restricts the resulting programs to a call-by-value discipline. In practice, this is not a difficulty since the resulting programs are usually evaluated using call-by-value anyway. The same problem arises, for example, with assignment conversion in the CPS transformation used by the Orbit compiler [17].

Finally, note that we could easily modify other normal-order CPS transformations for laziness in a similar manner. In particular, the optimal transformation of Danvy and Filinski [7] could be so modified.

6. Lazy CPS as Graph Reduction

We are now in a position to compare lazy CPS and graph reduction. Doing so, we find a strong similarity the two, especially between lazy CPS and self-modifying graph reduction.

Consider first the following minor modification to the lazy CPS transformation in which the functionality of fun and app has been "pushed" into force.

$$
\begin{array}{lcl}
\mathrm{Node} & = & \mathrm{Fun\ of\ Value} - \mathrm{Node\ ref} \\
 & | & \mathrm{App\ of\ Node\ ref\ x\ Node\ ref}
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{fun}\, f & = & \mathbf{ref}\,(\mathbf{Fun}\, f) \\
\mathbf{app}\, m\, n & = & \mathbf{ref}\,(\mathbf{App}\,(m, n)) \\
\mathrm{force\ r} & = & \mathbf{case}\ !r\ \mathbf{of}\ \ \mathbf{Funf} \Rightarrow \mathbf{return}\, f \\
 & & \qquad\qquad \mathbf{App}\,(m, n) \Rightarrow \mathbf{force}\, m \circ \mathbf{arg}\, n \circ \mathrm{updater} \\
\mathbf{update}\, r\, k & = & \lambda v.\,(r\ :\ =\ \ \mathbf{Fun}\, v;\ k\, v)
\end{array}
$$

(Note that the Fun constructor appears in the update function only because the current language is limited to functional values. However, it is not difficult to extend the language with other types of values, such as integers or lists.)

A comparison with the code in Section 2 reveals a clear similarity to graph reduction. The same resemblance exists between lazy CPS programs and self-modifying graph reduction, as described in Section 4. The only substantial

$$
\begin{array}{lcl}
[x] & \Rightarrow & x \\
[\lambda x_1 \ldots x_n . M] & \Rightarrow & \mathbf{fun}_n\ (\lambda x_1 \ldots x_n . [M]) \\
[M\ M_1 \ldots M_n] & \Rightarrow & \mathbf{app}_n\ [M]\,[M_1] \ldots [M_n]
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{fun}_n\ f & = & \text{ref}\ (\mathbf{fix}\ f'.\ \lambda s.\ \mathbf{case}\ s\ \text{of}\ m_1 :: \ldots :: m_n :: s' \Rightarrow \text{force}\ (f\ m_1 \ldots m_n)\ s' \\
 & & \qquad\qquad \textit{otherwise} \Rightarrow \textit{retumf's)} \\
\mathbf{app}_n\ m\ m_1 \ldots m_n & = & \mathbf{fix}\ r.\ \mathbf{ref}\ (\lambda s.\ \mathbf{force}\ m\ (m_1 :: \ldots :: m_n :: \text{nil}) \circ \text{updaters}) \\
\text{force}\ m\ s\ k & = & !m\ s\ k \\
\mathbf{return}\ f\ s\ k & = & k\ (\textit{As'}.\ \lambda k'.\ f\ (s @ s')\ k') \\
\mathbf{update}\ r\ s\ k & = & \lambda v.\ (r := v;\ v\ s\ k)
\end{array}
$$

Figure 5: The lazy CPS transformation extended with a spine stack. Standard ML list notation is used for stacks: nil is the empty list, :: is right-associative **infix** cons, and @ is **infix** append. The case-statement in the **definition** of fun,, corresponds to the argument satisfaction check.

differences between the CPS programs and graph reduction are the absence of a spine stack, a different strategy for updates, and a more **permissive** treatment of functions. We address each of these points in turn.

The most glaring **difference** from the code in Sections 2 and 4 is the lack of a spine stack. In the-current framework, arguments are captured in continuations instead of on a stack. However, an examination of the way argument continuations occur in lazy CPS programs reveals a stack discipline, so it is sensible to use a concrete stack (the spine stack) to represent these **continuations.**[5] Figure 5 shows how this stack may be added to the lazy CPS transformation. **There** are several things to note about this stack-based transformation. The **first** is that the use of a stack allows an efficient implementation of curried functions that consumes all of a function's arguments at once, rather than one at a *time.* However, it *also requires an* additional test, *called the argument satisfaction check,* which determines if there are enough arguments on the stack for the given function. When this test fails, a closure is returned which contains the partial **stack.**

Another **significant** difference between the two models is the strategy used for updates. Simple graph reducers update the graph after every reduction (see for example the update that occurs in the code presented for the S **combinator).** Unfortunately, under this approach a node may be updated many times with intermediate results before being updated with its **final** value. A more efficient approach, which we adopt here, is to perform only the **final** update. The use of continuations makes it obvious when this should occur. Several other graph reducers adopt a similar strategy, notably the Spineless G-machine **[6]** and the Spineless **Tagless** G-machine **[13].** This update strategy also **simplifies** the treatment of the spine stack. In the current framework, an application stacks its argument but in the code presented earlier, the application node itself is stacked and the argument must be extracted separately when required. This is because the updates that appear in the combinator rewrite rules need to know the location in the graph that is to be overwritten. By separating the updates from the reductions, the need to stack the application node is obviated.

A final (minor) difference between the two models is that in the earlier model, functions were restricted to be combinators, while no such restriction appears in the current model. (This restriction greatly simplifies the treatment of the environment.) Note, however, that this is not a restriction upon source programs; arbitrary functions are allowed in the source programs but they are converted into combinator **form** via a separate pass known as lambda-lifting [**10, 11].** A similar phase appears in CPS-based compilers, where it is known as closure-conversion [**1],** but it is not shown here because it **typically** occurs ***after*** CPS-conversion.

[5] **The non-argument continuations also obey a stack discipline and may also be represented with a concrete stack, either interleaved with the argument stack or stored separately. However, we will not make use of this fact in this paper.**

7. Related Work

Having described the relationship between graph reduction and lazy CPS, we now place the two in the broader landscape of implementation techniques for lazy evaluation. Of course, such techniques may vary along such dimensions as the presence of a stack or whether functions are restricted to be combinators, but these issues are really irrelevant to the question of laziness. The key issues are the representation of thunks *(i.e.,* delayed expressions) and the mechanism for `Updates`.

Bloss *et* al. [5] describes two implementation techniques for lazy evaluation, called *closure mode* and *cell mode.* In closure mode, a thunk is simply a nullary closure *(i.e.,* a closure that takes no arguments) which is forced by *entering the* closure, that is, transferring control to the code of the closure. The environment of the closure contains a status flag that tells whether this thunk has been evaluated before. If so, the environment contains the value of the thunk; othenvise, the environment contains whatever information will be required to determine that value. The code checks the status flag and then either evaluates the associated expression or returns its value. When the expression is evaluated, the status flag is set and the value is written to the environment. Cell mode is similar except the status flag is exposed to outside perusal, and status-checking is made the responsibility of the force operation rather than the code within the closure. This strategy enables a set of optimizations whereby the status check is bypassed if a thunk is forced in a context where it is definitely known to be evaluated or unevaluated.

Peyton Jones [13] describes a variant of closure mode, *called self-updating mode, which* avoids status checks altogether. A thunk is again represented by a nullary closure, but the status of the thunk is not made explicit. Instead, the status is implicitly represented by the code within the closure. The thunk is initialized to contain code that evaluates *the* delayed expression *and then replaces itselfwith code that simply returns the result value.* Subsequent attempts to force the thunk thus immediately return the desired result. Usually the code within a closure is physically represented by a code pointer, so replacing the code is accomplished simply by modifying the code pointer However, if the code is small enough, it is efficient to represent the code directly in the closure rather than indirectly via a code pointer In this case, replacing the code is accomplished by physically overwriting it with new code. Such self-modifying approaches are described in [2] and [16].

Graph reduction may be seen as either cell mode or self-updating mode, depending on the representation of tags. In either case, graph nodes are simply closures while graph edges are the free variables in the environments of closures. Nodes also contain a tag that distinguishes between interior nodes and leaf nodes (and furthermore between the different kinds of leaf nodes such as combinators or integers). This tag serves essentially the same purpose as the status flag which determines if a thunk has been evaluated or not. Interior (application) nodes and leaf nodes correspond to unevaluated and evaluated thunks respectively. Graph reducers which represent the tag explicitly (and branch on its value) may be seen as implementations of cell mode, while so-called 'tagless" reducers [13] which represent the tag implicitly as code (or code pointers) correspond to self-updating mode. Self-modifying graph reducers of course fall into the latter category.

The CPS equivalent of a nullary closure is a closure that takes just a single argument, a continuation. It is natural to consider corresponding continuation-passing modes for each of the modes mentioned above. Wang [18] describes implementations of closure continuation-passing mode, cell continuation-passing mode, and self-updating continuation-passing mode using call/cc in Scheme. Josephs [15] presents a continuation-based denotational semantics for a lazy functional language, using cell continuation-passing mode. Both approaches use continuations, but not continuation-passing style. Lazy CPS, as presented in this paper, may be seen as a description of self-updating continuation-passing mode in the context of CPS.

8. Conclusions and Future Work

In this paper we have made an observation about the connections between self-modifying graph reduction, continuations, and continuation-passing style. There are a number of possible directions for further work.

It has been suggested to us that the presentation of the lazy CPS transformation might be better made through the use

of monads [20]. This seems like a good idea, though we have not yet worked out the details. An interesting approach would be to express Plotkin's transformation as a monad. Then, it might be possible to use monad transformers to capture the modifications that are required to obtain the lazy CPS transformation.

Another subject for future investigation is the use of the lazy CPS transformation in implementations of lazy programming languages. CPS has become an increasingly popular intermediate representation in compilers [24, 17, 1]. One possibility, which we are currently investigating, is using a single CPS-based compiler back-end to support front-ends for both lazy and eager languages (or even languages with both lazy and eager features). A preliminary investigation indicates that this framework is also convenient for expressing optimizations based on semantic analyses such as strictness analysis [21] and sharing analysis [9]. Our early experience with implementations of the techniques presented here gives us reason to believe that this approach is viable.

Acknowledgements

The authors wish to thank Mark Leone for many detailed and useful comments on earlier drafts of this paper.

References

[1] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. *In Sixteenth ACM Symposium on Principles of Programming Languages,* pages 293–302, 1989.

[2] A. Augusteijn and G. vander Hoeven. Combinatorgraphs as self-reducing programs. Unpublished workshop presentation, 1984.

[3] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, 1987.

[4] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier, New York, 1981.

[5] Adrienne Bloss, Paul Hudak, and Jonathan *Young. Code* optimizations for lazy evaluation. *Lisp and Symbolic Computation,* 1:147–164, 1988.

[6] Geoffrey L. Burn, Simon L. Peyton Jones, and John D. Robson. The Spineless G-Machine. *In Proceedings of the ACM Conference on Lisp and Functional Programming, Snowbird,* pages 244–258, 1988.

[7] Olivier Danvy and Andrzej Filinski. Representing control. Technical Report TR-CS-91-2, Department of Computing and Information Sciences, Kansas State University, February 1991.

[8] Jan Fairbaim and Stuart Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In Gilles *Kahn,* editor, *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland,* pages 34-45. Springer-Verlag, 1987.

[9] Benjamin Goldberg. Detecting sharing of partial applications in functional programs. In Gilles Kahn, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland,* pages 408-425. Springer-Verlag, 1987.

[10] John Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Oxford University, 1983.

[1 1] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy,* pages 190-203. Springer-Verlag, September 1985.

[12] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, 1987.

[13] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. To appear *in* the *Journal of Functional Programming,* January 1992.

[14] Simon L. Peyton Jones and J. Salkild. The Spineless Tagless G-machine. *In The Fourth International Conference on Functional Programming Languages and Computer Architecture, London,* pages 184–201, September 1989.

[15] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science, 68:* 105-l 11, 1989.

[16] Philip J. Koopman, Peter Lee, and Daniel Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems,* 14(2):265–297, April 1992.

[17] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. *In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* pages 219–233, July 1986.

[18] *Ching* lin Wang. Obtaining *lazy* evaluation *with* continuations *in* Scheme. *Information Processing Letters,* 35:93–97, 1990.

[19] Robin Milner, *Mads* Tofte, and Robert Harper. *The Definition of Standard ML. The* MIT *Press,* Cambridge, Massachusetts, 1990.

[20] Eugenio Moggi. Computational lambda-calculus and monads. *In Symposium on Logic in Computer Science.* IEEE, June 1989.

[21] *A.* Mycroft. The *theory* and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming,* volume 83 of *LNCS,* pages 269-28 1. Springer-Verlag, 1980.

[22] Gordon D. Plotkin. Call-by-name, call-by-value and the *X-calculus. Theoretical Computer Science,* 1:125–159, 1975.

[23] John *C.* Reynolds. Definitional interpreters for higher-order programming languages. *In Proceedings of the ACM National Conference, New York, New York,* pages 7 17–740, 1972.

[24] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, 1978.

[25] David A. Turner A new implementation technique for applicative languages. *Software — Practice and Experience,* 9(1):3 l-49, January 1979.