combo: A Demo for Combinator Reduction

Revision 2

1. Introduction

This is the source to a free demo for a Lambda Calculus to combinator convertor and combinator reduction algorithm. The software in its present form dates from 1993 October 23, though it had been occasionally pursued as part of an on-going project in 1983-4 and 1987-8. It is released to the public domain free of any restrictions. You may use it for any purpose as you see fit. Both the extensional and non-extensional algorithms are incorporated. Revision 2 incorporates the η-rule, which defines extensionality.

The description to follow is for the optimal combinator reduction process with extensionality. The architecture for the reduction process has been rigorously derived from an abstract machine called the SCD machine – which represents a normal order counterpart to the SECD-2 machine.

The unique feature of this software is that its reduction algorithm is optimal: no expression is ever stored more than once anywhere at any time during the entire history of a session, no expression is ever reduced more than once, no evaluation is ever done that does not need to be done. Nothing is ever forgotten, nothing ever thrown away, therefore there is no garbage collection. In place of garbage collection is the optimal sharing and reuse of expressions – a strategy I call *Garbage Recycling*. The evaluation strategy is called: *Crazy Evaluation*. It is also possible to extend this strategy to add in a loop checker in order to stop infinite cycles. This has been done, in this implementation.

If one allows the dynamic creation of new combinator rules, one can go even further than even this. This is outlined below.

1.1. History

The first version was written in BASIC (it never got done), the second version in Pascal – my first Pascal program in fact, the third version in C – one of my first C programs, and was upgraded once in 1990, and a second time to this.

The conception underlying this software essentially dates from 1985 – as part of a project to develop a new programming paradigm based on combinatory logic … this was still before Functional Programming existed as anything more than a premonition; and long before I ever heard of any such thing. The λ-calculus to combinator conversion dates from a manual-algorithm that I've used since 1983.

The reduction algorithm actually conforming to the description below was added in 1994 September 28. The description below related to a feature (optimal reduction) which was an unintended consequence that simply emerged out of nowhere. The recommendation I made in the "SECD-2..." series was used in this design: the algorithm was rigorously derived from the specification of the SCD machine given below.

The cycle-checking was a consequent part of this upgrade; and is illustrated in one of the sample input files.

This additional effort culminated in a relatively minor but significant revision on 1995 February 19 when the algorithm was made extensional and the project became formally linked on 1995 February 24-25 with my ongoing SECD-2 project, as the definition was cast in terms of an abstract architecture that is for all intents and purposes the normal order form of the SECD-2 machine – called the SCD machine.

The SCD machine is an environment-free variation of SECD-2 that processes expressions in Normal order. The reason for the absence of the E component is that all the substitutions are processed at "compile-time". There are in effect two levels of processing going on which I've termed the α-*level* (corresponding to "compile-time") and β-*level* (corresponding to "run-time"). One interesting feature of the architecture is that even though the expressions (\x.e) f and (x = f, e) are equivalent and will always yield exactly the same results, the first is processed at the β-level and the second is processed at the α-level and a λ-term is never seen by the time the β-level is reached. This is where one is seeing the beginnings of a distinction emerging between the functional and imperative styles – not as distinctions between language families but as distinctions in the way terms are read and processed.

Future versions of SCD will blur the distinctions between the levels by allowing them to be interleaved in time and the functional/imperative distinction by allowing "virtual side-effects", a term I coined to reflect the fact that these side-effects arise purely as epiphenomena from the way name-clashes and substitutions are processed.

1.2. Compilation

The source is written in ANSI-C, and requires an ANSI-C/C++ compiler capable of supporting C99 or more recent. There are two compile-time switches at the head of the program, defined as follows:

- STATISTICS: 1 – show the number of reductions itemized by combinator; 0 – suppress the statistics
- SHOW_STEPS: 1 – show the combinator reductions performed; 0 – suppress the reduction output
- EXTENSIONAL: 1 – use the extensional reduction algorithm; 0 – define the F combinator

They are all set to 1 in the source.

The following files have been included:

- the source code in combo.c
- the reference files in combo.pdf or combo.doc
- the input files in in*.txt

2. User Interface

2.1. Command Line

The command line has the following format

combo

Inputs and outputs come directly through standard input and output, either interactively or through file redirection.

2.2. Input format

Sample input has been provided in the files "in*.txt". Each input is a single expression that consists of the following items

- *Combinators*: **I**, **K**, **D**, **T**, **W**, **U**, **B**, **C**, **S**, **F**
- *Identifiers*: variables and symbolic constants of the following forms
  - *C identifiers* (**a-z**,**A-Z**,_, followed by zero or more **0-9**,**a-z**,**A-Z**,_)
  - *Quoted identifiers*, enclosed in double quotes (**"**).
- *Function applications*: in the traditional format, the combination **f x** is used to denote the function **f(x)**. Applications associate to the left, e.g. **f x y = (f x) y**.
- *Lambda abstractions*: **\x y ... z . e** is used to denote **λx.(λy.**...**(λz.e)**...**))**. The dot is only necessary to remove ambiguity, otherwise optional.
- *Substitutions*: **x = f, e** equivalent to **(λx.e)f**.
- *Grouping*: parentheses may be used to explicitly group subexpressions.

2.3. Output format

Each lambda abstraction in the input is converted into an equivalent combinator expression. The underlying algorithm is $O(n^2)$, optimal compilation can be done by other methods in O(n log n) time. The resulting combinator expression is displayed on the first line in the form

Expression**:**

On the following lines, each of the combinator reductions performed is illustrated in the following form:

Redex ➔ Contractum

The final two lines will show the expression resulting from evaluation (the Normal Form), and statistics on the number of steps taken, in the following form

Normal Form Expression

steps: 3 (I 0, K 0, D 1, T 1, W 0, U 0, B 1, C 0, S 0, F 0)

The number of reduction steps taken is listed followed by a list itemized for each combinator.

Expressions are printed out with the same syntax as accepted for input. Sharing is represented by a series of assignments preceding the final expression. For example, the expression

x (x (S x) y) (y (x (S x) y))

will be printed out as

_0 = x (S x) y, x _0 (y _0)

The variable names **_0**, **_1**, **_2**, ... are reserved by the interpreter for this purpose, and should not be used in the input.

3. From Lambda Calculus to Combinatory Logic

A more rigorous approach to the λ-calculus (particularly the handling of substitution), shows how the combinator logic emerges almost as a necessary consequence. The β-equivalence rule can be cast in the form

$$\beta\colon\ (\lambda x.e)f = (x = f, e)$$

the α-equivalence rule in the form

$$\alpha\colon\ \lambda x.e = \lambda z.(x = z, e)$$

and the η-equivalence rule in the form

$$\eta\colon\ \lambda z.(az) = a$$

where z is a variable which does not occur freely in a.

A self-contained account, however, requires that one formulate an explicit definition for the substitution

$$(x = f, e)$$

as well. This may be done recursively over the structure of a term as follows

$$\sigma_I\colon\ (x = f, x) = f$$

$$\sigma_K\colon\ (x = f, y) = y \text{, for variables } y \neq x$$

$$\sigma_S\colon\ (x = f, tu) = (x = f, t)(x = f, u)$$

$$\sigma_\lambda\colon\ (x = f, \lambda y.e) = \lambda z.(x = f, y = z, e)$$

where the last equivalence applies with any variable $z \neq x, y$ that does not occur freely in e or f. Together, such explicit rules for substitution comprise what may be termed the σ-equivalences.

The σ_λ rule may be split into different subcases to include the following

$$(x = f, \lambda x.e) = \lambda x.e$$

$(x = f, \lambda y.e) = \lambda y.(x = f, e)$, for any other variable $y \neq x$ that does not occur freely e.

However, these rules may be derived, with the aid of the other rules; particularly, α-equivalence. To make the equivalence "deterministic" requires a means of canonically assigning new variables z. However, for merely posing a definition of equivalence, it does not matter.

On account of this consideration, however, it is more natural to keep the rules merely as equivalences, rather than as definitions, with substitution being an operator.

However, in that case, one may dispense with the substitution operator entirely, treating β-equivalence as a definition! Then, the α-rule actually becomes a special case of the η-rule

$$\lambda x.e = \lambda z.(\lambda x.e)z\ .$$

The σ-rules take on the following forms

$$\beta_I\colon\ (\lambda x.x)f = f$$

$$\beta_K\colon\ (\lambda x.y)f = y \text{, for variables } y \neq x$$

$$\beta_S\colon\ (\lambda x.tu)f = ((\lambda x.t)f)((\lambda x.u)f)$$

$$\beta_\lambda\colon\ (\lambda x.\lambda y.e)f = \lambda z.(\lambda x.(\lambda y.e)z)f$$

with the properties

$$\lambda x.(\lambda x.e)f = \lambda x.e$$

$$(\lambda x.\lambda y.e)f = \lambda y.(\lambda x.e)f$$

where, again, the last rule applies to all variables $y \neq x$ that do not occur freely e.

This leads naturally to the definition

$$\lambda x.x = \mathbf{I}$$

$$\lambda x.y = \mathbf{K}y \text{ , for variables } y \neq x$$
$$\lambda x.(tu) = \mathbf{S}(\lambda x.t)(\lambda x.u)$$

with the restated equivalences

$$\beta_I\text{: } \mathbf{I}x = x$$
$$\beta_K\text{: } (\mathbf{K}x)y = x \text{ , for variables } y \neq x$$
$$\beta_S\text{: } (\mathbf{S}xy)z = (xz)(yz)$$

The last rule, β_λ, leads to the further equivalences that define extensionality, such as $\mathbf{SK} = \mathbf{KI}$, $\mathbf{I} = \mathbf{SKK}$, $\mathbf{S}(\mathbf{K}x)(\mathbf{K}y) = \mathbf{K}(xy)$, $\mathbf{S}(\mathbf{K}x)\mathbf{I} = x$, and so on.

4. Compilation Method

Assignments are processed at compile time in a way consistent both with their reading as assignment statements and their reading as lambda expressions with the following equivalence

$$\mathbf{x} = f, e \leftrightarrow (\mathbf{\backslash x}.e)f$$

but still evaluated in normal order, which means *e* first (the order may be switched to *f*-first for assignments in future versions of the software).

Denoting the result of translating a Lambda expression, e, with substitution, σ, by <σ,e> the following rules apply

- <σ,x> = σ(x) for all identifiers and combinators, x
- <σ,f x> = <σ,f><σ,x>
- <σ,(x = e_1, e_2)> = <(σ,x←<σ,e_1>),e_2>
- <σ,(e)> = <σ,e>
- <σ,\$x_1$ x_2 ... x_n.y> = <σ,\$x_1$\$x_2$ ... \$x_n$.y>
- <σ,\x.e> = [x]<σ,e>

where (σ, x←e) is the substitution σ modified as follows

$$(\sigma, x{\leftarrow}e)(x) = e$$
$$(\sigma, x{\leftarrow}e)(y) = \sigma(y) \text{ if } y{\neq}x.$$

These rules are applied in a bottom-up manner. The symbol table is initially set to the identity substitution, I, given by

$$I(x) = x$$

The abstraction process, [*x*]*e*, is carried out according to the following rules. The expressions *a* and *b* denote expressions in which no *x* occurs, and *u* and *v* denote expressions in which at least one *x* occurs.

[x]x = **I**	[x]a = **K** a	[x](x x) = **D**	[x](a x) = a	[x](x b) = **T** b
[x](u x) = **W** [x]u	[x](x v) = **U** [x]v	[x](a v) = **B** a [x]v	[x](u b) = **C** [x]u b	[x](u v) = **S** [x]u [x]v

These rules are applied in a top-down manner.

Example 1: Head=(\x.x K), Pair=(\x y z.z x y), Head (Pair m n)

First, evaluating the subterm (\x.x **K**), we get

$$[x](x\ \mathbf{K}) = \mathbf{T}\ \mathbf{K}$$

Second, for the subterm (\x y z.z x y), we get

$$[x][y][z]\ (z\ x\ y) = [x][y]\ (\mathbf{C}\ [z](z\ x)\ y) = [x][y]\ (\mathbf{C}\ (\mathbf{T}\ x)\ y) = [x]\ \mathbf{C}\ (\mathbf{T}\ x) = \mathbf{B}\ \mathbf{C}\ [x](\mathbf{T}\ x) = \mathbf{B}\ \mathbf{C}\ \mathbf{T}$$

Combining these results into the main term, we get

<I, (Head=(\x.x **K**), Pair=(\x y z.z x y), Head (Pair m n))>=<(I,Head←**T K,** Pair←**B C T)**, Head (Pair m n)> = **T K** (**B C T** m n)

It is entirely possible to define abstraction solely in terms of **S**, **K** and **I**, with **I** being reducible by the equivalence **I** = **SKK**. This is done as follows:

$$[x]x = \mathbf{SKK}$$
$$[x]y = \mathbf{K}\ y, \text{ where } y{\neq}x$$
$$[x](e_1\ e_2) = \mathbf{S}\ [x]e_1\ [x]e_2$$

The first definition may then be regarded as an optimization in which the following equivalences are applied

SKK = **I**	**S**(**K**a)(**K**b) = **K**(ab)	**SII** = **D**
S(**K**a)**I** = a	**SI**(**K**b) = **T**b	**Sa**I = **W**a
SIb = **U**b	**S**(**K**a)b = **B**ab	**S**a(**K**b) = **C**ab

Example 2:
The following

x = (x = S x, (T = S, S = K, K = T, K S x (x = x x, S x))), K x

is equivalent to

K (S K (S x) (K (S x (S x))))

and reduces to

K (K (S x (S x)))

in 2 steps (+ 3 extensional steps).

Example 3:
This expression

x = K, x = S x, y = x S, S = S S, x S y

is

S K (S S) (S K S)

and reduces to **I** in 2 steps + 2 extensional steps or to **S K S** in 2 steps with the non-extensional algorithm.

Example 4:

s = K, K = S K I, I = S, S = s, S (S K I) Cryptic

is equivalent to

K (K (S K I) S) Cryptic

and also reduces to **I** in 2 steps + 2 extensional steps; or to **S K I** in 2 steps with the non-extensional algorithm.

Example 5:
If you have a lot of memory available, you may want to try and see what happens when you run the interpreter on the input

S (S S) S (S (S S) S) (S (S S) S)

which may be more compactly written as

M = S (S S) S, M M M

Its resolution is unknown. In fact, even **M M** *A* behaves somewhat chaotically for different selections of *A*

- **M M x** has a long and tortuous reduction sequence, if **x** is a variable
- **M M K** has a cyclic infinite reduction
- **M M C** has an infinite reduction

5. Implementing Iteration

The **F** combinator comes in handy in representing Primitive Recursive functions. One application (but by no means the only one) is in implementing a function iteration. Define

$$n(0) = \mathbf{K}\ \mathbf{I},\ n(u + 1) = \mathbf{F}\ n(u),\ u \geq 0$$

Then the combinator sequence n(k) f x will reduce to $f^k(x)$.

Consider the following system

$$s(0) = b,\ s(u + 1) = a\ s(u)$$

Its solution can be expressed in terms of n(u) by: s(u) = n(u) a b, or if we allow n(u) to stand for the number u, we can define s itself: s = \u(u a b) which is equivalent to s = **B C T** a b.

Consider the system

$$t(0) = c,\ t(u + 1) = s(u)\ t(u)$$

Here we have the following equivalences

$$t(0) = c = \mathbf{K}\ (\mathbf{K}\ \mathbf{I})\ a\ b\ c$$
$$t(1) = b\ c = \mathbf{K}\ \mathbf{I}\ a\ b\ c$$
$$t(2) = a\ b\ (b\ c) = \mathbf{F}\ a\ b\ c$$
$$t(3) = a\ (a\ b)\ (a\ b\ (b\ c)) = \mathbf{F}\ a\ (a\ b)\ (b\ c) = \mathbf{F}\ \mathbf{F}\ a\ b\ (b\ c) = \mathbf{F}\ (\mathbf{F}\ \mathbf{F}\ a)\ b\ c = \mathbf{B}\ \mathbf{F}\ (\mathbf{F}\ \mathbf{F})\ a\ b\ c$$

Define the functions

$$w(0) = \mathbf{K}\ (\mathbf{K}\ \mathbf{I}),\ w(u + 1) = \mathbf{B}\ \mathbf{F}\ (\mathbf{F}\ w(u))$$

Note that w(1) a b c = **K I** a b c, and w(2) a b c = **F** a b c. Then the general solution to the system above can be expressed as

$$t(u) = w(u)\ a\ b\ c$$

Since w(u + 1) can be expressed equivalently as

$$w(u + 1) = \mathbf{B}\ (\mathbf{B}\ \mathbf{F})\ \mathbf{F}\ w(u)$$

it follows that when we use n(u) to denote the number u that

$$w(u) = u\ \ (\mathbf{B}\ (\mathbf{B}\ \mathbf{F})\ \mathbf{F})\ \ (\mathbf{K}\ (\mathbf{K}\ \mathbf{I}))$$

Thus

$$t(u) = u\ (\mathbf{B}\ (\mathbf{B}\ \mathbf{F})\ \mathbf{F})\ (\mathbf{K}\ (\mathbf{K}\ \mathbf{I}))\ a\ b\ c$$

This is illustrated by the input file, *in5*, and resulting output, *out5*. The number of reduction steps is 10 + 5u (25 when u = 3) – the initial 10 steps consisting of reductions by 4 **C**'s, 3 **K**'s, 2 **I**'s, and a **T**, and the 5u steps consisting of 3u **F**'s and 2u **B**'s.

The intimate relation between this **F** combinator and the recursion process is demonstrated by the thresholding phenomenon where **F**, **F F**, **F F F**, **F F F F** and even **F F F F** x y z will all terminate but **F F F F F** will apparently keep on reducing forever in an unpredictably chaotic manner. None of the other combinators here has that property.

6. Reduction Method

The following reduction rules may be applied to the following combinations whenever they occur

I x → x	**K** x y → x	**W** x y → x y y	**T** x y → y x	**D** x → x x
U x y → y (x y)	**B** x y z → x (y z)	**C** x y z → x z y	**S** x y z → x z (y z)	**F** x y z → x y (y z)

The combination on the left and right of each rule is respectively called a *redex* and *contractum*.

The method used is optimal because it has the following features

- No reduction is performed twice – ever.
- No subexpression is ever stored twice.
- The least number of reductions logically possible are done.
- All cyclic infinite reduction sequences are recognized (but are blocked in the current version)
- Reduction is in normal order
- Every term with a normal form reduces to its normal form in a finite number of steps
- Every two beta-equivalent terms reduce to the same normal form if either has a normal form.

This is all subject to the memory limitations of the host processor. Apart from the equivalences of combinator expressions arising from extensionality (e.g. **S K x = I**) and the testing for strong normal form, the reduction algorithm performs the fewest possible reduction steps to arrive at normal form. For instance, the expression

S K ((\x.x x) (\x.x x))

will reduce to **I**, even though the bracketed subexpression has no normal form (*x* is the notation used by the software for the λ*x* operator). As another example, the expression

C C C C C C C C ....

will reduce to **C C C** with *only one* combinator reduction (**C C C C** → **C C C**) and 2 extensional steps to deduce that **C C C** is in strong-normal form ... regardless of how many **C**'s there are. Not even cyclic reductions are repeated. This method could be extended to output Rational Infinite Lambda terms in a finite number of steps, but this version of the interpreter will just halt. For instance, the expression

W D (W D)

would yield the Rational Infinite expression

... (**W D**) (**W D**) (**W D**).

The specific method is based directly on the following definitions for normal order reduction

- arity(**x**) = ∞, for all variables **x**
- arity(**I**) = arity(**D**) = 1
- arity(**K**) = arity(**W**) = arity(**T**) = arity(**U**) = 2
- arity(**B**) = arity(**C**) = arity(**S**) = arity(**F**) = 3
- arity(f x) = arity(f) – 1

where ∞ - 1 = ∞ > 0. Define the following

- T is in head-normal form ↔ arity(T) > 0

- T is a redex $\leftrightarrow$ arity(T) = 0
- T is an application $\leftrightarrow$ arity(T) < 0

Normal form cannot be easily defined syntatically. However, a set of sufficient conditions for a term, T, to be in normal form can be stated as follows

- x is in normal form, for all variables x.
- c is in normal form, for all combinators c.
- T U is in normal form if T and U are each in normal form and either arity(T) = ∞ or arity(U) = ∞.
- If T is in normal form then so is any abstraction of it, [x]T.

Common subexpression elimination is constantly performed by recycling subterms, so that no new term is ever created twice. When a new subterm is created, its arity will be set there and then, which determines its status once and for all. Any time that a new term needs to be made, either before or during reduction, it will first be determined if it already exists. If not, then it will be created and its arity will be set at that time. So the arity of a term are set once and for all. Its normal form status is tentatively set based on the observations above.

The abstraction algorithm used must ensure consistency with the requirement that combinators be regarded as normal form. Here, for instance, this is accomplished by

[y][**K** x y] = [y]x = **K** x
[x][**K** x] = [x](**K** x) = **K**

Other normal form terms could be inferred, e.g. **K** n is always in normal form for any normal form term n. The term (**U** n) is in normal form for any normal form term n, except for the cases **U I** → **D**, and **U** (**K** a) → **T** a. Things get more complicated beyond this, for instance

C (B B S) K → C	**B S K → B**	**S K → K I**
S T → W	**C T, W K, B I → I**	**B W K, B C C → I**
B (B f g) h → B f (B g h)	**B f I, B I f → f**	**B (K I) f → K I**

In fact, one can see by the last four examples that there's an entire monoidal algebra hidden within the combinators and within that a group subalgebra consisting of all the "reversible" combinators. So because things can get complicated, the determination of normal forms (even **K** n and **U** n) is deferred to run-time except for the four simple conditions noted above. I've actually tried to formulate finite axiom sets for extensional reduction with respect to different abstraction algorithms, but despite the elegance of some of the formulae (like those shown above) things can get pretty hairy.

The reduction process may be defined by the following two functions

- [T] = the result of reducing T to normal form
- <T> = the result of reducing T to head-normal form

where

- [T] = T if T is in normal form.
- [T U] = [T] [U] if arity(T) = ∞
- [H] = [x][H x] if 0 < arity(H) < ∞, where [x][H x] denotes the abstraction of [H x] by x.
- [R] = [C] if R is a redex and C its contractum.
- [T U] = [<T> U] if T U is an application

and

- <H> = H if H is in head-normal form.
- <R> = <C> if R is a redex and C its contractum.
- <T U> = <<T> U> if T U is an application.

A reduction sequence implementing this specification will always consist of a sequence of "lateral" steps from a redex to a contractum and "vertical" steps from an application to one of its subterms. If any term along this reduction sequence is encountered twice, then that proves the occurrence of a cyclic term. The sequence ends there. Otherwise, if the sequence ends in either normal form (for [T] reductions) or head-normal form (for <H> reductions) then all the lateral steps leading to it are traced back until the latest vertical step is reached. At this point, the vertical step is retraced up, and reduction continues on this term (which must be an application term), after using the result of the reduction of the subterm just exited from. If the original vertical step involved an application by a new variable, the reverse step will carry out the corresponding abstraction. This implements the clause [H] = [x][H x] above.

There is no guarantee that a reduction sequence will end in a cyclic or (head)-normal form. It is possible for an infinite number of distinct terms to be generated, but shortly before this occurs the interpreter's memory will be exhausted. As far as the interpreter is concerned, this term's evaluation status is undecidable.

During the reduction process, the state of a term is given by the following

state(T)	T's reduction state.
0	not currently being reduced
1	being reduced to head-normal form
2	being reduced to normal form
3	reduction to head-normal form suspended pending the reduction of a subterm
4	reduction to normal form suspended pending the reduction of a subterm
5	being reduced to [x][T x].

The auxillary link(T) is defined as follows

- for state(T) > 0: link(T) traces out the reduction path backwards from T
- for state(T) = 0: link(T) = [T] if currently known, <T> otherwise, if it is currently known, 0 else

Finally, normal(T) ≠ 0 ↔ T is known to be a normal form term.

When a new term T is being formed, besides initializing arity(T) the other auxilliary items are also initialized as follows

state(T) = 0
link(T) = T, if T is in head-normal form.

The function normal(T) is initialized to 0, except if

- T = x, a variable
- T = (U V), if U V is not a redex; normal(U) = normal(V) = 1; and either arity(U) = ∞ or arity(V) = ∞
- T = [x][U], where U is known to be in normal form.

When the interpreter is evaluating a term, the only new terms are those of the form: T U. Before this term is formed, the links from T and U will be traversed if necessary – which will never involve tracing more than two links each for T and U. This ensures that newly created terms take full advantage of any prior reductions done on their subterms.

7. The SCD Machine

7.1. The SCD Configuration

A configuration in the abstract machine takes on the following form

S,C:D

where

S = the current state
C = the current term
D = the Dump

The current state is defined by

S	Condition
s	starting reduction of [C]
t	starting reduction of <C>
e	reduction is complete

The Dump is a list of the form

State:Term:State:Term...:State:Term

of 0, 1, 2 or more pairs of the form (State,Term) where State = 1, 2, 3, 4 or 5.

In fact the entire purpose of the functions state(T) and link(T) is to embody this architecture, with the following correspondences

- state(T) > 0 ↔ (state(T),T) is an item in the Dump
- state(T) > 0 → link(T) points to the next item in the Dump
- If any term on the Dump is encountered a second time, this proves a cyclic infinite reduction exists. In addition, because reduction is in normal order, it proves that *every* reduction sequence of the current term is infinite.

This architecture is actually the normal-order counterpart to the SECD-2 machine in which the substitution environment has been eliminated due to the conversion to combinators.

The abstract architecture can then be rigorously derived from the following assumptions

- In **s**,C:D and **t**,C:D, state(C) == 0
- In **e**,C:D, state(C) == 0 and link(C) is the resulting (head-)normal form.

This architecture uses combinator reduction and abstraction to perform extensional beta reduction. Every term is reduced to (*strong*) *normal form*, which is defined in terms of the abstraction algorithm used.

The extensional version of reduction carries out reductions to strong normal forms. When necessary, the algorithm appends new dummy variables, reduces the result as far as possible and then abstracts out the variables using the same abstraction algorithm used during compilation. Therefore, by definition, a term T is considered to be in strong normal form if and only if

- T is in normal form
- the result of reducing T x to normal form and abstracting out the x yields T, where x is an otherwise unused variable.

7.2. The SCD Architecture

With these preliminaries set, we may then enumerate the elements of the abstract machine.

Rule 0 C → s,C

Rule 1 **Start State**

Rule 1a (Head)-normal forms

- t,H:D → e,H:D, if H is in head-normal form
- s,N:D → e,N:D, if N is in normal form

Rule 1b If link(R) == C or R is a redex with contractum C

- t,R:D → t,C:1:R:D
- s,R:D → s,C:2:R:D

except where link(R) == R which occurs in state s when R is in head-normal form)

Rule 1c If T U is an application

- t,(T U):D → t,T:3:(T U):D
- s,(T U):D → t,T:4:(T U):D

Rule 1d If T U is in head-normal form – applied in the following order

- s,H:D → s,(H x):5:H:D, if arity(H) < ∞, where x is a new variable
- s,(T U):D → s,T:4:(T U):D, T not in normal form
- s,(T U):D → s,U:4:(T U):D, T in normal form, U not
- s,(T U):D → e,(T U):D, if T, U are in normal form; mark (T U) as normal form and set link(T U) = (T U).

Rule 2 **End State**

Rule 2a **e**,C → halt and return C

Rule 2b **e**,C:1:P:D → **e**,C:D, set link(P) = C

Rule 2c **e**,C:2:P:D → **e**,C:D, set link(P) = C

Rule 2d The first applicable rule in the following is applied:

- **e**,C:3:(T U):s:(link(T) link(U)):D → **t**,(T U):s:(link(T) link(U)):D
- **e**,C:3:(T U):D → **t**,(link(T) link(U)):1:(T U):D

Rule 2e The first applicable rule in the following is applied:

- **e**,C:4:(T U):s:(link(T) link(U):D → **s**,(T U):s:(link(T) link(U)):D
- **e**,C:4:(T U):D → **s**,(link(T) link(U)):2:(T U):D

Rule 2f The first applicable rule in the following is applied; x is the variable used in the corresponding application of **Rule 1d**:

- **e**,C:5:H:s:[x]H:D → **e**, H:s:[x]H:D
- **e**,C:5:H:D → **e**,[x]C:2:H:D

7.3. The Reduction Algorithm

Let C and D denote the current and previous term in the reduction sequence (D being the first term in the Dump), and let S denote the current state, with S = 1 or 2 corresponding to states **t** and **s**. Then the following C-like algorithm defines the reduction method used:

Reduce(C):
Initialization
    S = 2, D = 0; /* *Rule 0* */
Start State
(A) *Lateral Step*
if C already has a link or if it's a redex:
    Reduction Step
    if C has no link, **then** N = C's contractum; **else** N = link(C);
    state(C) = S, link(C) = D, D = C, C = N;
    goto *Start State*

(B) *Vertical Step*
if C is an application:
    Initiate reduction of the head subterm to head-normal form
    state(C) = S + 2, link(C) = D, D = C, C = head(C), S = 1;
    goto *Start State*
else C is in head-normal form, but not in normal form and S == 2
    if arity(C) < ∞
    Application Step
        state(C) = 5, link(C) = D, D = C
        C = C x, where x is a new variable
        goto *Start State;*
    else if head(C) or tail(C) is not in normal form
    Initiate reduction of the subterms to normal form
        state(C) = S + 2, link(C) = D, D = C;
        if C is not yet in normal form **then** C = head(C); **else** C = tail(C);
        goto *Start State;*
    else if head(C) and tail(C) are both in normal form
    A new normal form has been found
        normal(C) = 1, link(C) = C;
        goto *End State;*

(C) *Cyclic Terms*
if (state(C) > 0) **then** halt, indicating that C is a cyclic term
regula
(D) *Normal Forms*
    if (S == 2 && normal(C) == 1) **then goto** *End State*;
    if (S == 1 && C is in head-normal form) **then goto** *End State*;
    else goto *Lateral Step*

End State
(E) *Trace back a Lateral Step*
if state(D) is 1 or 2 **then**
    N = D, D = link(D), state(N) = 0, link(N) = C
    goto *End State;*

(F) *Trace back a Application Step*
if state(D) is 5, x is the last variable created in an *Application* step that hasn't yet been abstracted over.
    N = D, D = link(D), C = [x]C, link(N) = C, state(N) = 0
    The *End State* is retained because the abstraction [x]C is always in normal form since C is in normal form.
    goto *End State;*

(G) *Trace back a Vertical Step*
if state(D) is 3 or 4 **then**
S = state(D) = state(D) – 2;
if C == D **then** D = link(C), state(C) = 0, link(C) = 0
goto *Start State*

(H) *Trace back the initialization step*
if D == 0
The dump is empty. All done.
The result is C.

8. Possible Extensions

Note that none of the features described below are present in the current version of the software. They may be added in on a future date.

8.1. Higher-Order Optimized Reduction & Reduction by Unification

Suppose one were to allow the creation of new combinators dynamically. If done correctly, this could provide a large savings on the number of steps carried out. Instead of storing individual histories, essentially representing different instantiations of the same combinator rule – one is storing the entire rule itself in one place. Then ideally, one could add in another optimality feature

No composite combinator is executed in terms of more primitive steps more than once.

This involves nothing less than the full use of pattern unification. In this setup, free variables are now interpreted as logical variables. Any time a reduction is carried out, the resulting pattern is saved. Whenever a new term is being created, it will be checked against the set of existing terms by unification. If there's a generalization of it in the table that has a reduction, that reduction will be used as an initial link.

This also provides a natural basis for a garbage collection strategy. When two terms unify to a third, an appropriate deletion strategy can be taken when one turns out to be a generalization of the other term. In some cases it may even be possible to produce a simultaneous generalization of both terms (by pattern disunification) and store that in place of the other terms.

8.2. Strict Functions, Conditional Expressions

In order to incorporate built-in functions such as

A + B

which are typically strict in each of their arguments, or the conditional expression (written in a C-like syntax):

A? B: C

one needs to go back and directly modify the reduction rules themselves. From there, a corresponding adjustment will need to be made to the reduction method described above.

For instance, rules corresponding to A + B and A? B: C could be formed as follows

[A + B] = [A] + [B], i.e. evaluate [A] and [B], then add their results.
[A? B: C] = [B] if [A] = 0
[A? B: C] = [C] if [A] ≠ 0

The rules for head-normal reduction need not change, however. The rule for the conditional can be simplified by defining it in terms of the combinators

A? B: C = **if** A B C

where (**if** A) is a conversion function defined by

if true = **K**
if false = **K I**

Then one only needs to formulate a reduction rule for the if function, e.g.

[**if** A] = **if** [A]

There's one exception to head-normal reduction scheme. In order to get strict evaluation, one must impose the rule

<R> = [R], where arity(R) = 0

when R is headed by a strict function.

8.3. Direct Manipulation of Rational Infinite Lambda Terms

It would also be desirable to close off the reduction algorithm above by allowing cyclic reductions to rational infinite lambda terms to be represented. It would also be desirable to extend the compilation method to allow rational infinite lambda terms to be handled.

It turns out that this ties in very closely with the Unification feature alluded to above. One will not really be able to carry out either extension without the other.

One question naturally arises: what kind of notation does one use to represent infinite terms? Well, this is a case where the answer preceded the question by several years and is actually part of what inspired the question. Since a rational infinite lambda term is (by definition) a term containing a finite number of distinct subterms, up to isomorphism, you can represent such terms as a system of the following form

L_0: T_0
L_1: T_1
...
L_n: T_n

where T_0, T_1, ..., T_n may each contain one or more of the the labels L_0, L_1, ..., L_n. How does one distinguish an embedded label from a variable syntactically? With the **goto**!

For instance, assuming that the extensions in (b) are also present, the rational infinite lambda expression

x = 0, y = 1, x < n? (x = x + 1, y = y * 2, x < n? (x = x + 1, y = y * 2, [...]): y): y

would be represented by the system

L_0: x = 0, y = 1, **goto** L_1
L_1: x < n? (x = x + 1, y = y * 2, **goto** L_1): y

One can even introduce some syntactic sugaring to represent some of the more common structures, e.g.

while T_1 T_2() T_3

will stand for

L_0: T_1? T_2(**goto** L_0): T_3

where T_2() is a *context*: a term with a missing subterm (a "hole"). Thus, the term in this example could be represented by the syntax

```
x = 0, y = 1,
while (x < n) (x = x + 1, y = y * 2, ())
y
```

where the empty parenthesis pair () denotes the position of the "hole". This syntax is actually quite a bit more flexible than the analogous C-like syntax. One could, for instance, also write the expression (which, interestingly enough turns out to be equivalent to the one above)

```
x = 0, y = 1,
while (x < n) (x = x + 1, y = (), y * 2)
y
```

which represents the infinite term given by the syntax

L_0: x = 0; y = 1; **goto** L_1
L_1: x < n? (x = x + 1; y = **goto** L_1; y*2): y

which, when written out in full looks like this

x = 0, y = 1, x < n? (x = x + 1, y = x < n? (x = x + 1, y = [...], y*2): y, y*2): y