Efficient Bracket Abstraction

M. Randall Holmes

June 30, 2006

Contents

1 Coverage

This document is not a paper but notes for a paper. Much needs to be fleshed out further, and mistakes and infelicities may be present.

The early sections are a standard discussion of lambda-calculus and combinators (I use $(x \mapsto T)$ instead of $(\lambda x.T)$ and $f(x)$ instead of (fx)). There shouldn't be anything really novel in this discussion. The potentially novel discussion is under "efficient bracket abstraction".

2 Naive Theory of Functions

The naive theory of functions we learn in high school allows us to define functions using expressions containing variables as follows: $f(x) = 2x + 1$.

A problem with this is that we do not give permanent names to most functions: this can be fixed by introducing the notation $(x \mapsto T)$ as a name for the function f introduced by the definition $f(x) = T$. A usual notation for this in the formal study of this concept of function is $(\lambda x.T)$, but we will stick with the older notation to emphasize our reliance on the naivete of our readers.

An expression like $x+2y$ can be thought of as a function of two variables. One might expect the introduction of ordered pairs at this point, and a definition of $x + 2y$ as something like $((x, y) \mapsto x + 2y)$, as notation for a function f satisfying $f(x, y) = x + 2y$. but a different approach is more conceptually economical. Consider instead $(x \mapsto (y \mapsto x + 2y))$, which is a function of x which returns a function of y: this is the function g introduced by the definition $g(x)(y) = x + 2y$. Functions of multiple variables defined in this way are said to be "curried".

We introduce a notation for *substitution*, though we give fair warning that our aim is to eliminate substitution entirely. The notation $T[U/x]$, where T and U are terms and x is a variable, is intended to denote the result of substituting the term U for the variable x in the term T.

Of course one needs to be careful in defining substitution in the presence of the $(x \mapsto T)$ notation. For example, $(x \mapsto a)$ is the constant function whose value everywhere is a: $(x \mapsto a)[b/a]$ is naively $(x \mapsto b)$, and this is the constant function with value b everywhere, just as one would expect. However, $(x \mapsto a)[x/a]$ naively evaluates to $(x \mapsto x)$, which is the identity function, not the expected constant function with value x!

We know what the problem is: the variable x in $(x \mapsto T)$ is a dummy variable which doesn't really have reference in quite the normal sense. A substitution for x should not have any action inside $(x \mapsto T)$ and any substitution of a term U containing x for a variable appearing in T will have unexpected effects.

Both of these difficulties with the proper definition of substitution can be fixed using the same observation. The dummy variable in any function notation $(x \mapsto T)$ can be renamed: $(x \mapsto T) = (y \mapsto T[y/x])$, when y is a new variable not appearing at all in T.

We regiment our language: atomic terms are variables and such atomic constants as we may wish to introduce. For any terms T and U, $T(U)$ is a term, representing the application of T (as a function) to U. For any variable x and term T, $(x \mapsto T)$ is a term. For the moment, all terms are constructed in this way.

The usual presentation of terms of λ-calculus or combinatory logic represents the application of T to U by (TU), and then allows removal of parentheses from the left (so $((TU)V)$ can be written TUV, while $(T(UV))$is written $T(UV)$. It is amusing to observe that removing all parentheses which enclose terms not of the form $T(U)$ in our notation gives the simplified form of the standard notation.

We now define substitution $T[U/x]$ by induction on the construction of the

term T. If a is an atomic term, $a[U/x]$ is U if a is typographically the same as x, and a if a is different from x typographically. If T and U are terms, we define $T(U)[V/x]$ as $T[V/x](U[V/x])$. We define $(x \mapsto T)[U/y]$ as $(z \mapsto T[z/x][U/y])$, where z is the lexicographically first variable not appearing in T or U (though in fact any variable satisfying these conditions will do).

We can now state two basic rules for calculation with functions in terms of substitution.

We can rename "bound variables" (notice that we actually have not introduced any notion of free or bound variable): $(x \mapsto T) = (y \mapsto T[y/x])$ when y does not occur in T.

We can evaluate functions in accordance with our naive intention (but with a better chance of correct computation): $(x \mapsto T)(U) = T[U/x]$.

We note that the latter axiom, telling us how to define each function evaluation using substitution, can actually be used to define away substitution in terms of function evaluation: this is our eventual intention.

3 Substitution and Equality

We have just introduced the notion of equality. We now describe our rules for equational reasoning (which also depend on the notion of substitution, as we should expect).

reflexive: $T = T$ is a theorem.

symmetric: If $T = U$ is a theorem then $U = T$ is a theorem.

transitive: If $T = U$ is a theorem and $U = V$ is a theorem, then $T = V$ is a theorem.

substitution: If $U = V$ is a theorem, then $T[U/x] = T[V/x]$ is a theorem (for any term T and variable x).

generality: If $T = U$ is a theorem, then $T[V/x] = U[V/x]$ is a theorem, (for any term V and variable x).

We also accept occurrences of the axioms of renaming and evaluation as theorems. Our axiom of renaming is usually called the rule of α-reduction, and our axiom of evaluation is usually called the rule of β-reduction.

The rules above should appear quite natural, but notice that where the substitution rule is used to replace U with V where $U = V$ is a theorem in a context which binds some of the variables in U and V, our account of what is being done must be a little more complex, because terms containing bound variables do not have reference in quite the usual sense. One explanation is that we are using a weak form of the principle of extensionality.

4 Combinators and Inefficient Bracket Abstraction

We now commence the program of elimination of the notion of substitution in favor of the notion of evaluation.

We review the clauses in the definition of substitution:

$x[U/x] = U$

$a[U/x] = a$ when a is atomic and different from x.

$T(U)[V/x] = T[V/x](U[V/x])$

$(x \mapsto T)[U/y] = (z \mapsto T[z/x][U/y])$ where z does not occur in T or U.

We rephrase the first three entirely in terms of evaluation rather than substitution (recalling that $T[U/x] = (x \mapsto T)(U)$):

$(x \mapsto x)(U) = U$

$(x \mapsto a)(U) = a$ when a is atomic and different from x.

$(x \mapsto T(U))(V) = (x \mapsto T)(V)((x \mapsto U)(V))$

We introduce a constant I to represent $(x \mapsto x)$, the identity function which appears in the first clause. We can express $(x \mapsto a)$ as $K(a)$, where the constant K is defined as $(a \mapsto (x \mapsto a))$. More esoterically, we can express $(x \mapsto T)(V)((x \mapsto U)(V))$ as $S(x \mapsto T)(x \mapsto U)$, where $S = (f \mapsto (g \mapsto (x \mapsto f(x)(g(x)))))$.

For each term T containing no function notation $(x \mapsto U)$, define a term $[x]T$ containing neither function notation nor any occurrence of the variable x, such that $[x]T(x) = T$ is a theorem. The definition is recursive and the proof is an exactly parallel induction:

the variable: $[x]x = I$

other atoms: $[x]a = K(a)$

applications: $[x]T(U) = S([x]T)([x]U)$

This is the definition of (inefficient) bracket abstraction. It is straightforward to prove that $[x]T(U) = T[U/x]$ and moreover that $[x]T = (x \mapsto T)$ (this last because $[x]T$ is always actually equal to some $(x \mapsto T')$ (seen from the definitions of the combinators) and the rules of equality show that function notations with the same extension are equal): the variable binding notation for functions is eliminable.

The final rule of substitution needs to be recast in terms of bracket abstraction before function notation can actually be effectively eliminated:

original form: $(x \mapsto T)[U/y] = (z \mapsto T[z/x][U/y])$ where z does not occur in T or U.

evaluation form: $(y \mapsto (x \mapsto T))(U) = (z \mapsto (y \mapsto (x \mapsto T)(z))(U))$

bracket form: $([y][x]T)(U) = ([z][y]([x]T)(z))(U)$

It is possible to convert the last version into a finite number of concrete equational axioms which implement the weak extensionality implicit in substitution of equals for equals inside function notations, the *combinatory axioms* of Curry, but we will not do this here.

We refer to this as inefficient bracket abstraction because the blowup in size of terms in standard function notation can be exponential. Observe that each atomic term a in a function notation term is replaced by I if it is the binding variable, and otherwise by $K(a)$; each application $T(U)$ is replaced by a term $S(T')(U')$; in our notation, the blowup factor for any local part of a term in standard function notation is about 4^n if it lies inside n function subterms: the potential blowup is exponential in the size of the term.

The exponential blowup can be averted if we use rules of a slightly different kind. $[x]T$ can be taken to be $K(T)$ in case T does not contain the variable x: this can be enforced in the bracket abstraction algorithm by a postprocessing step which converts abstracts which would be of the form $S(K(U))(K(V))$ to $K(U(V))$. A related simplification makes $[x]T(x)$ equal to T instead of $S(K(T))(I)$ when T is itself an abstract or when the occurrence of T is in a suitable position inside an abstract. The effect of this simplification is to prevent the iterated constant factor of expansion of subterms which lie in many nested function subterms: the blowup, while still considerable in practice, is merely polynomial.

5 Implementations of basic concepts in our function calculus

It is a well-known observation that λ-calculus is a universal programming language (and of course so is combinatory logic). To see how this works, we note that the ordered pair $\langle x, y\rangle$ can be represented by $(z \mapsto z(x)(y))$: the projection operators π_1 and π_2 are represented as $(x \mapsto x(K(I)))$ and $(x \mapsto x(K))$, respectively. This allows construction of all sorts of finite data structures.

The natural number n is represented by the Church numeral $(f \mapsto (x \mapsto f^n(x)))$. 0 is $K(I)$ and for any Church numeral n, $n+1$ is equivalent to $(f \mapsto (x \mapsto (f(n(f)(x)))))$, so the successor operation is captured as $(n \mapsto (f \mapsto (x \mapsto (f(n(f)(x))))))$

Addition, multiplication, exponentiation, and the zero test on natural numbers are readily represented by combinators (we use the projection operators to code truth values). The use of projection operators as truth values makes it easy to code propositional logic operations.

Recursion is handled by the presence of fixed points for every operation: any function f has the fixed point $(x \mapsto f(x(x)))(x \mapsto f(x(x)))$. This is quite alarming (if we consider this as a standard theory of functions) but it enables recursive definitions adequate for computation.

It is straightforward to get all partial recursive functions represented as combinators. Further, one can use recursion on data structures built using pairs to get sensible representations of all commonly used data structures.

6 Efficient Bracket Abstraction

This section introduces a new algorithm for bracket abstraction. The motivation of the approach is to cause abstracts to be precisely parallel in structure to the terms from which they are abstracted, and to maintain this condition when abstraction is iterated.

Initially, we present this system with new operations (involving natural number indices), but it can in principle be presented in terms of a finite set of combinators (it can be used to produce relatively compact representations in terms of the set of combinators we have been using).

We present a modification of our formal language. Atomic terms are variables, such arbitrary constants as one might desire, and constants K^n (with K^0 being written I and K^1 being written K). For any term T and positive natural number n, $|T|^n$ is a term, with $|T|^1$ being written $|T|$. For any terms T and U, $T(U)^n$ is a term, with $T(U)^0$ being written $T(U)$.

We have the following axioms:

the variable: $I(x) = x$

other atomic terms: $|T|(U) = T$

constant functions of the variable: $K^n(x) = |x|^n$

constant function terms: $|T|^{n+1}(U) = |T|^n$

complex constant functions: $|T(U)^m|^n = |T|^n(|U|^n)^{m+n}$

iterated constant functions: $||T|^m|^n = |T|^{m+n}$

basic application terms: $T(U)^1(V) = T(V)(U(V))$

higher application terms: $T(U)^{n+1}(V) = T(V)(U(V))^n$

These axioms do not exhaust our axiom set; there will be further axioms governing higher-order applications. We could write $K^n(T)$ instead of $|T|^n$, but see the stratified combinatory logic below, where K^n is not a function (except for $n = 0$).

These axioms correspond to clauses of a bracket abstraction algorithm:

the variable: $[x]x = I$

other atomic terms: $[x]a = |a|$ when a is atomic and typographically distinct from x.

constant functions of the variable: $[x]|x|^n = K^n$

constant function terms: $[x]|a|^n = |a|^{n+1}$ when a is atomic and typographically distinct from x.

complex constant functions: $[x]|T(U)^m|^n = [x]|T|^n(|U|^n)^{m+n}$

iterated constant function: $[x]||T|^m|^n = [x]|T|^{m+n}$

basic application terms: $[x]T(U) = [x]T([x](U))^1$

higher application terms: $[x]T(U)^n = [x]T([x]U)^{n+1}$

An iterated abstraction from a term has structure precisely parallel to that of the parent term (after constant functions of complex terms are converted into complex terms involving constant functions of atomic terms using the complex constant functions axiom); bound variables are replaced by terms of the form $|K^m|^n$. The indices m and n are related to two schemes of deBruijn indexing. The size of the iterated abstraction is larger than the size of the original term by no more than a logarithmic factor (this blowup is caused by the increase in size of numerals, if these are supposed written in the usual way).

A further nice feature of this scheme of bracket abstraction is that it supports a sensible representation of substitution of equals for equals inside abstractions.

One could introduce a postprocessing phase converting terms $|T|^m(|U|^m)^{n+m}$ to $|T(U)^n|^m$ (restoring constant functions of complex terms where possible). This has the advantage of making the definition of substitution for complex terms inside erstwhile function notations work correctly.

We give axioms for computation of higher order abstractions:

the variable: $|I|^n(x)^n = x$

verification: True for $n = 0$. $|I|^{n+1}(x)^{n+1}(y) = |I|^n(x(y))^n = x(y)$ by inductive hypothesis; bracket abstracts with the same extension are expected to be equal.

other atomic terms: $|T|^{n+1}(U)^n = |T|^n$

verification: True for $n = 0$. $|T|^{n+2}(U)^{n+1}(y) = T^{n+1}(U(y))^n = |T|^n$ by inductive hypothesis, and a bracket abstract all of whose value are $|T|^n$ is expected to be $|T|^{n+1}$.

constant functions of the variable: $|K^n|^m(|x|^m)^m = |x|^{n+m}$

verification: True for $m = 0$. $|K^n|^{m+1}(|x|^{m+1})^{m+1}(y) = |K^n|^m(|x|^m)^m = |x|^{n+m}$ by inductive hypothesis. A bracket abstraction with all values $|x|^{n+m}$ is equal to $|x|^{n+m+1}$.

constant function terms: $|T|^{n+1+m}(U)^m = |T|^{n+m}$

verification: True for $m = 0$. $|T|^{n+2+m}(U)^{m+1}(y) = |T|^{n+1+m}(U(y))^m = |T|^{n+m}$ by induction; a bracket abstract all of whose values are $|T|^{n+m}$ is itself $|T|^{n+m+1}$

complex constant functions: $|T(U)^m|^n = |T|^n(|U|^n)^{m+n}$

verification: This is unaltered.

iterated constant functions: $||T|^m|^n = |T|^{m+n}$

verification: This is unaltered.

basic application terms: $T(U)^1(V) = T(V)(U(V)$

verification: This is unaltered (we can't usefully generalize this to a higher-level outer application).

higher application terms: $T(U)^{n+m+1}(V)^m = T(V)^m(U(V)^m)^{n+m}$

verification: This is true for $m = 0$. $T(U)^{n+m+2}(V)^{m+1}(y) = T(U)^{n+m+2}(y)(V(y))^m = T(y)(U(y))^{n+m+1}(V(y))^m = T(y)(V(y))^m(U(y)(V(y))^m)^{n+m} = T(V)^{m+1}(y)(U(V)^{m+1}(y))^{n+m} = T(V)^{m+1}(U(V)^{m+1})^{n+m+1}(y)$ establishing the point given that abstracts with the same extension are equal.

These rules allow evaluations inside function notations to be carried out inside the bracket abstractions representing these function notations.

There's more work to do here: a full reduction algorithm should be defined. I expect that a full "strong reduction" with nice properties can be defined here (though showing that the nice properties hold will probably require work). There are more details...

7 A compositional semantics for function notation

The combinators we use in efficient bracket abstraction can be understood directly by considering a specific way of giving meaning to function notations in terms of their subterms.

Each term in our language has a sequence of referents, determined by the number of function notations ($x \mapsto T$) of which it is a proper subterm.

The referent of any occurrence of a subterm U of a top-level term T is the standard referent of $(x_1 \mapsto (x_2 \mapsto (x_3 \mapsto \ldots (x_n \mapsto U)\ldots)))$, where the x_i's are the binders of the function notation subterms of T which contain this occurrence of U in the same nested order in which they appear in T. The previous sentence is the complete definition: the remarks which follow are consequences. The referent of an occurrence of a constant atomic subterm a inside n function subterms will be $|a|^n$. The referent of a variable x_i inside n binders will be an appropriate $|K^p|^q$, where $p+q = n-1$. Consideration of the meanings assigned to the three terms V, W, and $V(W)$ inside n binders reveals that if the referent

of V is V' and the referent of W is W', then the referent of $V(W)$ will be $V'(W')^n$. If all atomic subterms are replaced by their referents, all applications are replaced by the appropriate indexed application, and all binders are dropped completely, we obtain precisely our efficient bracket abstraction.

The referent of an occurrence of an atomic constant or of a free variable is its n-fold iterated constant function if it appears within n function notations. The referent of an occurrence of a bound variable is $|K^p|^q$, where there are p binderss appearing more deeply nested than its deepest occurrence as a binder above the occurrence (the one that counts) and q binders less deeply nested. Notice that either p or q could be used as a "de Bruijn index" for the variable (the usual scheme of deBruijn indexing uses p or $p+1$).

8 Eliminating the numerals

Because of the indexing, our system for efficient bracket abstraction appears to use infinitely many primitive operations. We show how we can at least in principle get the same effect while using a finite set of combinators.

We have seen above [this has not yet been written] how to represent numerals as combinators. A level of efficiency similar to that of the usual decimal or binary notation can be obtained as follows: use I to represent 1. The combinator 2 satisfies $2(f)(x) = f(f(x))$. The combinators E and O are used to attach additional binary digits: $O(n)(f)(x) = f(2(n(f))(x)); E(n)(f)(x) = 2(f)(2(n(f))(x))$. The effect of O is to double a Church numeral and add 1; the effect of E is to double a Church numeral and add 2; together, this allows a representation of natural numbers with logarithmic efficiency. There are probably better ways to represent the standard numerals and operations on them from the standpoint of actual evaluation.

Now $|T|^n$ can be represented as $n(K)(T')$ (where T' is the translation of T, and $T(U)^n$ can be represented as $n(S')(T')(U')$, where S' is Turner's combinator satisfying $S'(f)(g)(h)(x) = f(g(x))(h(x))$.

In order to efficiently implement our bracket abstraction algorithm using this finite representation, it would appear to be convenient to explicitly type terms representing numerals.

9 Relations to prior work

Of course most of what we have said about function notation and combinators parallels what is said in Curry and Feys, with different notation.

The combinator S' is found in Turner's algorithm for bracket abstraction, which is stated to have quadratic space efficiency. There are algorithms which are reported to have $n \log(n)$ space efficiency (as this one does) but they appear to be less intuitively appealing than this one. Healfdane Goguen's bracket abstraction algorithm is very close in spirit to this one and enjoys space efficiency (with close parallelism of structure) and straightforward simulation of

reductions inside function notations (λ-terms) for basically the same reasons that this system enjoys these advantages. He grades the combinators I, S, and K with numerical indices rather than grading the application operation itself (though he brifly discusses doing this he does not explore it). Goguen's interests are different from ours; he is interested in representing reasoning via the Curry-Howard isomorphism rather than in efficient representation of functions without bound variables.

10 Typed and stratified systems

It seems that the $n \log(n)$ space efficiency is lost in a typed system, because one will replace numerical indices, which can be made compact, with lists of types, which are incompressible in general. It appears that the space efficiency becomes quadratic (I haven't checked this). More efficiency could be achieved, perhaps, if type inference were used and many type indices could then be replaced with numerals.

For any type τ, there will be I^τ of type $\tau \to \tau$, an identity function. For any term T of type σ, there will be a constant function $|T|^\tau$ of type $\tau \to \sigma$ taking the value T at every argument of type τ. We can then see that the correct index on an iterated constant function is a list of types. The typing of analogues of our indexed applications is more complicated.

The author's combinatory logic TRC, which is equivalent to a stratified λ-calculus equivalent to a variant of Quine's set theory "New Foundations", has a more economical implementation along the lines of this paper: indeed, the treatment of stratified λ-calculus is just as efficient as that of untyped λ-calculus (equivalent to our scheme of function notation).

Stratified λ-calculus has variables and atomic constants as its atomic terms. If T and U are terms, $T(U)$ is a term. For certain choices of terms T and variables x, $(x \mapsto T)$ is a term: these are the terms for which the "relative type" of no occurrence of the variable x in T differs from 0. The relative type of the obvious occurrence of x in the term x is 0; if the relative type of an occurrence of x in T is n, the type of the analogous occurrence in $T(U)$ will be $n+1$, in $U(T)$ will be n, and in $(y \mapsto T)$ will be $n-1$ (if the latter term is well-formed). Note that relative types can be negative integers.

Stratified λ-calculus is equivalent to a combinatory logic introduced by the author. Atomic terms are variables, the constants `Abst` and `Id`, and whatever other atomic constants may be desired. If T and U are terms, then $|T|$ and $T(U)$ are terms. $|T|$ is the constant function with value T.

The following axioms are introduced to define the behaviour of the combinators:

identity: $\texttt{Id}(x) = x$

constant functions: $|x|(y) = x$

distribution: $\texttt{Abst}(f)(g)(x) = f(|x|)(g(x))$

complex constant functions: $|x(y)| = \texttt{Abst}(|x|)(|y|)$

This induces an inefficient bracket abstraction algorithm. Let x be any term containing no application subterm (atom or iterated constant function of an atomic term):

the "variable": $[x]x = \texttt{Id}$

other simple terms: $[x]a = |a|$ when a is a term other than x not containing any application subterm.

application: $[x]T(U) = \texttt{Abst}([|x|]T)([x]U)$ Note the occurrence of abstraction relative to $|x|$ instead of x!

complex constant function: $[x]||T(U)|^n = [x]||\texttt{Abst}(|T|)(|U|)|^{n-1}$ (here the numerical index is meta-notation).

It is straightforward to prove that if T is a term containing x with no relative type other than 0 (x contains x with relative type 0; if T contains x with relative type n, then $T(U)$ contains x with relative type $n+1$, $U(T)$ contains x with relative type n, and $|T|$ contains x with relative type $n-1$), then $[x]T$ does not contain x at all, contains any variable y with type $n-1$ iff T contains y with type n and y is distinct from x, and satisfies "$([x]T)(x) = T$ is a theorem". This further implies that every term of the stratified λ-calculus can be translated into a function-notation-free term.

This algorithm can be reduced from exponential blowup of terms of stratified λ-calculus to polynomial blowup by tricks similar to those used with untyped λ-calculus, but the blowup is considerable nonetheless (especially because of the need to "explode" iterated constant functions of complex terms, though these can often be "reimploded" after abstraction). I've done some actual experiments with these.

The efficient bracket abstraction introduced here adapts perfectly to this system.

We have the following axioms:

the variable: $\texttt{Id}(x) = x$

other atomic terms: $|T|(U) = T$

constant function terms: $|T|^{n+1}(U) = |T|^n$

complex constant functions: $|T(U)^m|^n = |T|^n(|U|^n)^{m+n}$

iterated constant functions: $||T|^m|^n = |T|^{m+n}$

basic application terms: $T(U)^1(V) = T(|V|)(U(V))$

higher application terms: $T(U)^{n+1}(V) = T(|V|)(U(V))^n$

Note that the axiom for K^n has disappeared, since K^n is not a function in stratified systems except when $n = 0$.

These axioms correspond to clauses of a bracket abstraction algorithm. x is a term containing no application subterm.

the variable: $[x]x = \texttt{Id}$

other atomic terms: $[x]a = |a|$ when a has no application subterm and typographically distinct from x.

complex constant functions: $[x]|T(U)^m|^n = [x]|T|^n(|U|^n)^{m+n}$

iterated constant function: $[x]||T|^m|^n = [x]|T|^{m+n}$

basic application terms: $[x]T(U) = [|x|]T([x](U))^1$ Note the occurrence of abstraction relative to $|x|$ instead of x!

higher application terms: $[x]T(U)^n = [|x|]T([x]U)^{n+1}$ Note the occurrence of abstraction relative to $|x|$ instead of x!

These axioms can presumably be generalized to higher applications just as above. (NOTE: check details on this). The abstraction algorithm presented here has the same level of efficiency for stratified λ-calculus that our original algorithm has for untyped λ-calculus: I'm going to test this in a reimplementation of the Watson theorem prover (which can be seen on my web site; I have already implemented a toy version of Watson using the Curry abstraction algorithm adapted to *TRC*).

TRC is interesting because it is more like typed λ-calculus than like untyped λ-calculus in logical respects: addition of a function representing equality with suitable properties (and of the axioms supporting substitution of equals for equals inside λ-terms) gives a system which has the same consistency strength and mathematical power as the usual typed λ-calculus, though it is untyped. Unlike the stratified λ-calculus with the same extensions, its definition does not directly refer to types.

It is possible to reduce to a finite set of combinators just as for the standard system above (use a compact representation of the numerical indices as combinators and an analogue $\texttt{Abst}'$ of S').

We have observed elsewhere that *TRC* could be regarded as fulfilling Curry's program for combinatory logic (with the technical refinement of stratification modifying his overambitious requirements for abstraction); the inefficiency of actually working with combinators vitiates this claim somewhat. However, this presentation of *TRC* will be essentially as easy to work with as stratified λ-calculus, at least for a computer.

11 A Computer Data Type for Terms

```
datatype Term =

     Variable of int*int | (* Variable(m,n) = |x_n|^m *)

     Constant of int*string  |  (* Constant(n,s) = |s|^n *)

     K of int*int | (* K(m,n) = |K^n|^m -- notice that K(n,0) = |I|^n *)

     App of int*Term*Term;  (* App(n,T,U) = T(U)^n *)
```

No explicit constant function construction is provided (or needed), since in abstracts one always pushes any application of the constant function operator down to iterated constant functions of atomic terms. If one wishes to represent a constant function of a complex term, one can explicitly use a K^n.

A term type for *TRC* would look similar. I might include two kinds of application and keep the possibility of using unstratified proper class functions.

Actual experiments are anticipated in a Watson reimplementation. I might also write a toy functional programming language and see how it works.