# Bottom-up $\beta$-reduction: uplinks and $\lambda$-DAGs

Olin Shivers
Georgia Institute of Technology
shivers@cc.gatech.edu

Mitchell Wand
Northeastern University
wand@ccs.neu.edu

**Abstract**

The $\lambda$-calculus is a simple language with far-reaching use in the programming-languages and formal-methods communities, where it is frequently employed to represent, among other objects, functional programs, formal proofs, and types drawn from sophisticated type systems. Of the three basic operations on terms in the $\lambda$-calculus—$\alpha$-conversion, $\beta$-reduction, and $\eta$-reduction—it is $\beta$-reduction that accomplishes the "heavy lifting" of term manipulation. (The other two operations are simple to implement.) Unfortunately, naïve implementations of $\beta$-reduction can lead to exponential time and space blowup.

This paper presents a new representation of $\lambda$-calculus terms that allows for fast, space-efficient operations on these structures. The representation is surprisingly simple, and is based on two ideas: (1) representing terms as a directed acyclic graph, allowing sharing, and (2) using explicit backpointers from children up to parents, allowing us to replace blind search with minimal, directed traversals.

We discuss the formal correctness of the algorithm, compare it with alternate techniques, and present comparative timings of implementations. An appendix contains a complete annotated code listing of the core data structures and algorithms, in Standard ML.

## 1 Introduction

The $\lambda$-calculus is a simple language, with an associated term-rewriting system [1]. There are only three forms in the basic language: $\lambda$ expressions, variable references, and applications of a function to an argument:

$$t \in \text{Term} ::= \lambda x.t \;\mid\; x \;\mid\; t_f\ t_a$$

where "$x$" stands for a member of some infinite set of variables. (We'll also allow ourselves parenthesisation of terms to indicate precedence in the parsing of concrete examples.)

$\beta$-reduction is the operation of taking an application term whose function subterm is a $\lambda$-expression, and substituting the argument term for occurrences of the $\lambda$'s bound variable in the function body. The result, called the *contractum,* can be used in place of the original application, called the *redex*. We write

$$(\lambda x.b)\, a \;\;\Rightarrow\;\; [x \mapsto a]b$$

to express the idea that the redex applying function $\lambda x.b$ to argument $a$ reduces to the contractum $[x \mapsto a]b$—by which we mean term $b$, with free occurrences of $x$ replaced with term $a$.

We can define the core substitution function with a simple recursion:

$$\begin{array}{lll}
[y \mapsto t][\![x]\!] & = t & x = y \\
[y \mapsto t][\![x]\!] & = x & x \neq y \\
[x \mapsto t][\![t_f\ t_a]\!] & = ([x \mapsto t]t_f)([x \mapsto t]t_a) & \\
[x \mapsto t][\![\lambda y.b]\!] & = \lambda y'.([x \mapsto t][y \mapsto y']b) & y' \text{ fresh in } b \text{ and } t.
\end{array}$$

Note that, in the final case above, when we substitute a term $t$ under a $\lambda$-expression $\lambda y.b$, we must first replace the $\lambda$-expression's variable $y$ with a fresh, unused variable $y'$ to ensure that any occurrence of $y$ in $t$ isn't "captured" by the $[x \mapsto t]$ substitution. If we know that there are no free occurrences of $y$ in $t$, this step is unnecessary—which is the case if we adopt the convention that every $\lambda$-expression binds a unique variable.

It is a straightforward matter to translate the recursive substitution function defined above into a recursive procedure:

```
fun subst(term, var, val)
  if term is variable then
    if term = var then val
    else term
  else if term is application then {
    new := NewAppNode()
    new.fun := subst(term.fun,var,val)
    new.arg := subst(term.arg,var,val)
    return new
    }
  else { // term is lambda
    new := NewLambdaNode()
    new.var := NewVar()
    body1 := subst(term.body, term.var, new.var)
    new.body := subst(body1, var, val)
    return new
    }
```

Consider the case of performing a substitution $[y \mapsto t]$ on an application $t_f\ t_a$. Our procedure will recurse on both subterms of the application... but we could also use a less positive term in place of "recurse" to indicate the trouble with the algorithmic handling of this case: search. In the case of an application, the procedure will blindly search *both* subterms, even though one or both may have no occurrences of the variable for which we search. Suppose, for example, that the function subterm $t_f$, is very large—perhaps millions of nodes—but contains no occurrences of the substituted variable $y$. The recursive substitution will needlessly search out the entire subterm, constructing an identical copy of $t_f$. What we want is some way to direct our recursion so that we don't waste time searching into subterms that do not contain occurrences of the variable being replaced.

```
Procedure addItem(node, i)
  if node = nil then {
      new := NewNode()
      new.val   := i
      new.left  := nil
      new.right := nil
      }
  else if node.val < i then {
      new := NewNode()
      new.right := addItem(node.right,i)
      new.left  := node.left
      new.val   := node.val
      }
  else if node.val > i then {
      new := NewNode()
      new.left  := addItem(node.left,i)
      new.right := node.right
      new.val   := node.val
      }
  else new := node
  return new
  }
```

Figure 1: Make a copy of ordered binary tree *node*, with added entry *i*. The original tree is not altered.

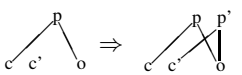
## 2 Guided tree substitution

Let's turn to a simpler task to develop some intuition. Consider an integer set kept as an ordered binary tree (figure 1). There are three things about this simple algorithm worth noting:

- **No search**
  The pleasant property of ordered binary trees is that we have enough information as we recurse down into the tree to proceed only into subtrees that require copying.
- **Steer down; build up**
  The algorithm's recursive control structure splits decision-making and the actual work of tree construction: the downward recursion makes the decisions about which nodes need to be copied, and the upward return path assembles the new tree.
- **Shared structure**
  We copy only nodes along the spine leading from targeted node to the root; the result tree shares as much structure as possible with the original tree.

## 3 Guiding tree search with uplinks

Unfortunately, in the case of $\beta$-reduction, there's no simple, compact way of determining, as we recurse downwards into a tree, which way to go at application nodes—an application has two children, and we might need to recurse into one, the other, both, or neither. Suppose, however, that we represent our tree using not only down-links that allow us to go from a parent to its children, but also with redundant up-links that allow us to go from a child to its parent. If we can (easily) find the leaf node in the original tree we wish to replace, we can chase uplinks along the spine from the old leaf to the tree root, copying as we go (figure 2). This gives us the algorithm of figure 3, presented somewhat abstractly for simple binary trees. The core iteration of this algorithm is the $c \mapsto c'$ upcopy:



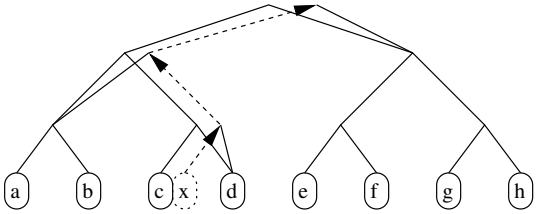




Figure 2: Replacing a single leaf in a binary tree by following uplinks. Here, we make a copy of the original tree, replacing leaf $c$ with $x$. Dotted lines show the path of algorithm as it copies up the spine of the tree from $c$ to the root.

```
while old ≠ root do {
  newparent := NewNode()
  if old is left-child of parent then {
      newparent.left  := new
      newparent.right := old.right
      }
  else {
      newparent.right := new
      newparent.left  := old.left
      }
  old := old.parent
  new := newparent
  }
```

Figure 3: Copy a tree with child→parent uplinks, replacing leaf *old* with *new*.

We take a child $c$ and its intended replacement $c'$, and replicate the parent $p$ of $c$, making the $c \mapsto c'$ substitution. This produces freshly-created node $p'$; we can now iterate, doing a $p \mapsto p'$ upcopy into the parent of $p$ at the next step, and so on, moving up through the original tree until we reach the root.

Note the similar properties this upcopy algorithm has with the previous algorithm: no search required; we build as we move upwards; we share as much structure as possible with the old tree, copying only the nodes along the "spine" leading from the leaf back to the root. For a balanced tree, the amount of copying is logarithmic in the total number of nodes. If we can somehow get our hands on the leaf node to be replaced in the old tree, the construction phase just follows uplinks to the root, instead of using a path saved in the recursion stack by the downwards search.

## 4 Upcopy with DAGs

We can avoid space blowup when performing $\beta$-reduction on $\lambda$-calculus terms if we can represent them as directed acyclic graphs (DAGs), not trees. Allowing sharing means that when we substitute a large term for a variable that has five or six references inside its binding $\lambda$-expression, we don't have to create five or six distinct copies of the term (that is, one for each place it occurs in the result). We can just have five or six references to the same term. This has the potential to provide logarithmic compression on the simple representation of $\lambda$-calculus terms as trees. These term DAGs can be thought of as essentially a space-saving way to represent term trees, so we can require them, like trees, to have a single top or root node, from which all other nodes can be reached.

When we shift from trees to DAGs, however, our simple functional upcopy algorithm no longer suffices: we have to deal with the fact that there may be multiple paths from a leaf node (a variable

```
Procedure upcopy(childcopy, parent, relation)
  if parent.cache is empty then {
      parcopy := NewNode()
      if relation is "left child" then
      then {
          parcopy.left  := childcopy
          parcopy.right := parent.right
          }
      else {
          parcopy.right := childcopy
          parcopy.left  := parent.left
          }
      parent.cache := parcopy
      for-each <grandp,gprel> in parent.uplinks do
        upcopy(parcopy, grandp, gprel)
      }
  else {
      parcopy := parent.cache
      if relation is "left child"
      then parcopy.left  := childcopy
      else parcopy.right := childcopy
      }
```

Figure 4: Procedure upcopy makes a copy of a binary DAG, replacing the *relation* child (left or right) of *parent* with *childcopy*.

reference) of our DAG up to the root of the DAG. That is, any term can have multiple parents. However, we can modify our upwards-copying algorithm in the standard way one operates on DAGs: we search upwards along all possible paths, marking nodes as we encounter them. The first time we copy up into a node $n$, we replicate it, as in the previous tree algorithm, and continue propagating the copy operation up the tree to the (possibly multiple) parents of $n$. However, before we move upwards from $n$, we first store the copy $n'$ away in a "cache" field of $n$. If we later copy up into $n$ via its other child, the presence of the copy $n'$ in the cache slot of $n$ will signal the algorithm that it should not make a second copy of $n$, and should not proceed upwards from $n$—that has already been handled. Instead, it mutates the existing copy, $n'$, and returns immediately.

The code to copy a binary DAG, replacing a single leaf, is shown in figure 4. Every node in the DAG maintains a set of its uplinks; each uplink is represented as a $\langle \textit{parent}, \textit{relation} \rangle$ pair. For example, if node $c$ is the left child of node $p$, then the pair $\langle p, \text{left-child} \rangle$ will be one of the elements in $c$'s uplink set.

The upcopy algorithm explores each edge on all the paths from the root of the DAG to the copied leaf exactly once; marking parent nodes by depositing copies in their cache slots prevents the algorithm from redundant exploration. Hence this graph-marking algorithm runs in time proportional to the number of edges, *not* the number of paths (which can be exponential in the number of edges). Were we to "unfold" the DAG into its equivalent tree, we would realise this exponential blowup in the size of the tree, and, consequently, also in the time to operate upon it. Note that, analogously to the tree-copying algorithm, the new DAG shares as much structure as possible with the old DAG, only copying nodes along the spine (in the DAG case, spines) from the copied leaf to the root.

After an upcopy has been performed, we can fetch the result DAG from the cache slot of the original DAG's root. We must then do another upwards search along the same paths to clear out the cache fields of the original nodes that were copied, thus resetting the DAG for future upcopy operations. This cache-clearing pass, again, takes time linear in the number of edges occurring on the paths from the copied leaf to the root:

```
Procedure clear(node)
  if node.cache = nil then return
  node.cache := nil
  for-each <par,rel> ∈ node.uplinks do
    clear(par)
```

## 5 Operating on λ-DAGs

We now have the core idea of our DAG-based $\beta$-reduction algorithm in place, and can fill in the details specific to our $\lambda$-expression domain.

**Basic representation** We will represent a $\lambda$-calculus term as a rooted DAG.

**Sharing** Sharing will be generally allowed, and sharing will be *required* of variable-reference terms. That is, any given variable will have no more than one node in the DAG representing it. If one variable is referenced by (is the child of) multiple parent nodes in the graph, these nodes will simply all contain pointers to the same data structure.

**Bound-variable short-cuts** Every $\lambda$-expression node will, in addition to having a reference to its body node, also have a reference to the variable node that it binds. This, of course, is how we navigate directly to the leaf node to replace when we begin the upcopy for a $\beta$-reduction operation. Note that this amounts to an $\alpha$-uniqueness condition—we require that every $\lambda$-expression bind a unique variable.

**Cache fields** Every application node has a cache field that may either be empty or contain another application node. $\lambda$-expression nodes do not need cache fields—they only have one child (the body of the $\lambda$-expression), so the upcopy algorithm can only copy up through a $\lambda$-expression once during a $\beta$-reduction.

**Uplinks** Uplinks are represented by $\langle \textit{parent}, \textit{relation} \rangle$ pairs, where the three possible relations are "$\lambda$ body," "application function," and "application argument." For example, if a node $n$ has an uplink $\langle l, \lambda\text{-body} \rangle$, then $l$ is a $\lambda$-expression, and $n$ is its body.

**Copying λ-expressions** With all the above structure in place, the algorithm takes shape. To perform a $\beta$-reduction of redex $(\lambda x.b)\, a$, where $b$ and $a$ are arbitrary subterms, we simply initiate an $x \mapsto a$ upcopy. This will copy up through all the paths connecting top node $b$ and leaf node $x$, building a copy of the DAG with $a$ in place of $x$, just as we desire.

Application nodes, having two children, are handled just as binary-tree nodes in the general DAG-copy algorithm discussed earlier: copy, cache & continue on the first visit; mutate the cached copy on a second visit. $\lambda$-expression nodes, however, require different treatment. Suppose, while we are in the midst of performing the reduction above, we find ourselves performing a $c \mapsto c'$ upcopy, for some internal node $c$, into a $\lambda$ parent of $c$: $\lambda y.c$. The general structure of the algorithm calls for us to make a copy of the $\lambda$-expression, with body $c'$. But we must also allocate a fresh variable, $y'$, for our new $\lambda$-expression, since we require all $\lambda$-expressions to bind distinct variables. This gives us $\lambda y'.c'$. Unfortunately, if old body $c$ contains references to $y$, these will also occur in $c'$—not $y'$. We can be sure $c'$ contains no references to $y'$, since $y'$ was created after $c'$! We need to fix up body $c'$ by replacing all its references to $y$ with references to $y'$.

Luckily, we already have the mechanism to do this: before progressing upwards to the parents of $\lambda y.c$, we simply initiate a $y \mapsto y'$ upcopy through the existing DAG. This upcopy will proceed along

the paths leading from the $y$ reference, up through the DAG, to the $\lambda y.c$ node. If there are such paths, they *must* terminate on a previously-copied application node, at which point the upcopy algorithm will mutate the cached copy and return.

Why must these paths all terminate on some previously copied application node? Because we have already traversed a path from $x$ up to $\lambda y.c$, copying and caching as we went. Any path upwards from the $y$ reference must eventually encounter $\lambda y.c$, as well—this is guaranteed by lexical scope. The two paths must, then, converge on a common application node—the only nodes that have two children. That node was copied and cached by the original $x$-to-$\lambda y.c$ traversal.

When the $y \mapsto y'$ upcopy finishes updating the new DAG structure and returns, the algorithm resumes processing the original $c \mapsto c'$ upcopy, whose next step is to proceed upwards with a $(\lambda y.c) \mapsto (\lambda y'.c')$ upcopy to all of the parents of $\lambda y.c$, secure that the $c'$ sub-DAG is now correct.

**The single-DAG requirement** We've glossed over a limitation of the uplink representation, which is that a certain kind of sharing is not allowed: after a $\beta$-reduction, the original redex must die. That is, the model we have is that we start with a $\lambda$-calculus term, represented as a DAG. We choose a redex node somewhere within this DAG, reduce it, and *alter the original DAG to replace the redex with the contractum*. When done, the original term has been changed—where the redex used to be, we now find the contractum. What we *can't* do is to choose a redex, reduce it, and then continue to refer to the redex or maintain an original, unreduced copy of the tree. Contracting a redex kills the redex; the term data structure is not "functional" in the sense of the old values being unchanged. (As we discuss later, we can, however, "clone" a multiply-referenced redex, splitting the parents between the original and the clone, and then contract only one of the redexes.)

This limitation is due to the presence of the uplinks. They mean that a subterm can belong to only one rooted DAG, in much the same way that the backpointers in a doubly-linked list mean that a list element can belong to only one list (unlike a singly-linked list, where multiple lists can share a common tail). The upcopy algorithm assumes that the uplinks exactly mirror the parent→child downlinks, and traces up through all of them. This rules out the possibility of having a node belong to multiple distinct rooted DAGs, such as a "before" and "after" pair related by the $\beta$-reduction of some redex occurring within the "before" term.

Hence the algorithm, once it has finished the copying phase, takes the final step of disconnecting the redex from its parents, and replacing it with the contractum. The redex application node is now considered dead, since it has no parents, and can be removed from the parent/uplink sets of its children and deallocated. Should one of its two children thus have its parent set become empty, it, too, can be removed from the parent sets of its children and deallocated, and so forth. Thus we follow our upwards-recursive construction phase with a downwards-recursive deallocation phase.

It's important to stress, again, that this deallocation phase is not optional. A dead node must be removed from the parent sets of its children, lest we subsequently waste time doing an upcopy from a child up into a dead parent during a later reduction.

**Termination and the top application** Another detail we've not yet treated is termination of the upcopy phase. One way to handle this is simply to check as we move up through the DAG to see if we've arrived at the $\lambda$-expression being reduced, at which point we could save away the new term in some location and return without further upward copying. But there is an alternate way to handle this. Suppose we are contracting redex$(\lambda x.b)\, n$, for arbitrary subterms $b$ and $n$. At the beginning of the reduction operation, we first check to see if $x$ has no references (an easy check: is its uplink set empty?). If so, the answer is $b$; we are done.

Otherwise, we begin at the $\lambda$-expression being reduced and scan downwards from $\lambda$-expression to body, until we encounter a non-$\lambda$-expression node—either a variable or an application. If we halt at a variable, it *must* be $x$—otherwise $x$ would have no references, and we've already ruled that out. This case can also be handled easily: we simply scan back through this chain of nested $\lambda$-expressions, wrapping fresh $\lambda$-expressions around $n$ as we go.

Finally, we arrive at the general case: the downward scan halts at the topmost application node $a$ of sub-term $b$. We make an identical copy $a'$ of $a$, *i.e.* one that shares both the function and argument children, and install $a'$ in the cache slot of $a$.

Now we can initiate an $x \mapsto n$ upcopy, knowing that all upwards copying must terminate on a previously-copied application node. This is guaranteed by the critical, key invariant of the DAG: all paths from a variable reference upward to the root *must* encounter the $\lambda$-node binding that variable—this is simply lexical-scoping in the DAG context. The presence of $a'$ in the cache slot of $a$ will prevent upward copying from proceeding above $a$. Node $a$ acts as a sentinel for the search; we can eliminate the root check from the upcopy code, for time savings.

When the upcopy phase finishes, we pass $a'$ back up through the nested chain of $\lambda$-expressions leading from $a$ back to the top $\lambda x.b$ term. As we pass back up through each $\lambda$-expression $\lambda y.t$, we allocate a fresh $\lambda$-expression term and a fresh variable $y'$ to wrap around the value $t'$ passed up, then perform a $y \mapsto y'$ upcopy to fix up any variable references in the new body, and then pass the freshly-created $\lambda y'.t'$ term on up the chain.

(Note that the extended example shown in section 7 omits this technique to simplify the presentation.)

## 6 Fine points

These fine points of the algorithm can be skipped on a first reading.

**Representing uplinks** A node keeps its uplinks chained together in a doubly-linked list, which allows us to remove an uplink from a node's uplink set in constant time. We will need to do this, for example, when we mutate a previously copied node $n$ to change one of its children—the old child's uplink to $n$ must be removed from its uplink set.

We simplify the allocation of uplinks by observing that each parent node has a fixed number of uplinks pointing to it: two in the case of an application and one in the case of a ł-expression. Therefore, we allocate the uplink nodes along with the parent, and thread the doubly-linked uplink lists through these pre-allocated nodes.

An uplink doubly-linked list element *appears* in the uplink list of the child, but the element *belongs* to the parent. For example, when we allocate a new application node, we simultaneously allocate two uplink items: one for the function-child uplink to the application, and one for the argument-child uplink to the application. These three data structures have identical lifetimes; the uplinks live as long as the parent node they reference. We stash them in fields of the application node for convenient retrieval as needed. When we mutate the application node to change one of its children, we also shift the corresponding uplink structure from the old child's uplink list to the new child's uplink list, thus keeping the uplink pointer information consistent with the downlink pointer information.

**The single-reference fast path** Consider a redex $(\lambda x.b)\, n$, where the $\lambda$-expression being reduced has exactly one parent. We know what that parent must be: the redex application itself. This application node is about to die, when all references to it in the

term DAG are replaced by references to the contractum. So the $\lambda$-expression itself is about to become completely parentless—*i.e.*, it, too, is about to die. This means that any node on a path from $x$ up to the $\lambda$-expression will also die. Again, this is the key invariant provided by lexical scope: all paths from a variable reference upward to the root *must* encounter the $\lambda$-expression binding that variable. So if the $\lambda$-expression has no parents, then all paths upwards from its variable must terminate at the $\lambda$-expression itself.

This opens up the possibility of an alternate, fast way to produce the contractum: when the $\lambda$-expression being reduced has only one parent, mutate the $\lambda$-expression's body, altering all of $x$'s parents to refer instead to $n$. We do no copying at all, and may immediately take the $\lambda$-expression's body as our answer, discarding the $\lambda$-expression and its variable $x$ (in general, a $\lambda$-expression and its variable are always allocated and deallocated together).

**Opportunistic iteration** The algorithm can be implemented so that when a node is sequencing through its list of uplinks, performing a recursive upcopy on each one, the final upcopy can be done with a tail recursion (or, if coded in a language like C, as a straight iteration). This means that when there is no sharing of nodes by parents, the algorithm tends to iteratively zip up chains of single-parent links without pushing stack frames.

## 7 Extended example

We can see the sequences of steps taken by the algorithm on a complete example in figure 5. Part 5(a) shows the initial redex, which is $(\lambda x.(x(\lambda y.x(uy)))(\lambda y.x(uy)))\ t$, where the $(\lambda y.x(uy))$ subterm is shared, and $t$ and $u$ are arbitrary, unspecified subterms. To help motivate the point of the algorithm, imagine that the subterms $t$ and $u$ are enormous—things we'd like to avoid copying or searching—and that the $\lambda x$ node has other parents besides application 1—so we cannot blindly mutate it at will, without corrupting what the other parents see.

In the following subfigure, 5(b), we focus in on the body of the $\lambda$-expression being reduced. We iterate over the parents of its variable-reference $x$, doing an $x \mapsto t$ upcopy; this is the redex-mandated substitution that kicks off the entire reduction. The first parent of $x$ is application 3, which is copied, producing application $3'$, which has function child $t$ instead of the variable reference $x$, but has the same argument child as the original application 3, namely the $\lambda y$ term. Dotted lines show new DAG structure. The copy $3'$ is saved away in 3's cache slot, in case we upcopy into 3 from its argument child in the future.

Once we've made a copy of a parent node, we must recursively perform an upcopy for it. That is, we propagate a $3 \mapsto 3'$ upcopy to the parents of application 3. There is only one such parent, application 2. In subfigure 5(c), we see the result of this upcopy: the application $2'$ is created, with function child $3'$ instead of 3; the argument child, $\lambda y$, is carried over from the original application 2. Again, application $2'$ is saved away in the cache slot of application 2.

Application 2 is the root of the upcopy DAG, so once it has been copied, control returns to application 3 and its $3 \mapsto 3'$ upcopy. Application 3 has only one parent, so it is done. Control returns to $x$ and its $x \mapsto t$ upcopy, which proceeds to propagate upwards to the second parent of $x$, application 4.

We see the result of copying application 4 in subfigure 5(d). The new node is $4'$, which has function child $t$ where 4 has $x$; $4'$ shares its argument child, application 5, with application 4. Once again, the copy $4'$ is saved away in the cache slot of application 4.

Having copied application 4, we recursively trigger a $4 \mapsto 4'$ upcopy, which proceeds upwards to the sole parent of application 4. We make a copy of $\lambda y$, allocating a fresh variable $y'$, with the new body $4'$. This is shown in subfigure 5(e).

Since the new $\lambda y'$ term binds a fresh variable, while processing the $\lambda y$ term we must recursively trigger a $y \mapsto y'$ upcopy, which begins in subfigure 5(f). We iterate through the parents of variable-reference $y$, of which there is only one: application 5. This is copied, mapping child $y$ to replacement $y'$ and sharing function child $u$. The result, $5'$, is saved away in the cache slot of application 5.

We then recursively trigger a $5 \mapsto 5'$ upcopy through the parents of application 5; there is only one, application 4. Upon examining this parent (subfigure 5(g)), we discover that 4 already has a copy, $4'$, occupying its cache slot. Rather than create a second, new copy of 4, we simply mutate the existing copy so that its argument child is the new term $5'$. Mutating rather than freshly allocating means the upcopy proceeds no further; responsibility for proceeding upwards from 4 was handled by the thread of computation that first encountered it and created $4'$. So control returns to application 5, which has no more parents, and then to $y$, who also has no more parents, so control finally returns to the $\lambda y$ term that kicked off the $y \mapsto y'$ copy back in subfigure 5(f).

In subfigure 5(h), the $\lambda y$ term, having produced its copy $\lambda y'$, continues the upcopy by iterating across its parents, recursively doing a $\lambda y \mapsto \lambda y'$ upcopy. The first such parent is application 3, which has already been copied, so it simply mutates its copy to have argument child $\lambda y'$ and returns immediately.

The second parent is application 2, which is handled in exactly the same way in subfigure 5(i). The $\lambda y$ term has no more parents, so it returns control to application 4, who has no more parents, and so returns control to variable reference $x$. Since $x$ has no more parents, we are done. The answer is application $2'$, which is shown in subfigure 5(j). We can change all references to application 1 in the DAG to point, instead, to application $2'$, and then deallocate 1. Depending on whether or not the children of application 1 have other parents in the DAG, they may also be eligible for deallocation. This is easily performed with a downwards deallocation pass, removing dead nodes from the parent lists of their children, and then recursing if any child thus becomes completely parentless.

## 8 Formal specification and correctness

In this section, we will more formally specify the core of the bottom-up reduction algorithm and outline the basic structure of our proof of its correctness. A precise, fully-detailed argument is beyond the scope of this paper; we provide only the structural skeleton, eliding details and simplifying the formalisations, and rely on the reader to interpolate where necessary. We proceed by defining the essential core of the algorithm as a pair of math functions, about which we can reason; this definition strips away details of the data-structures (such as the use of doubly-linked lists, *etc.*) in favor of using simple mathematical structures such as sets, graphs and partial functions.

To begin, let us define our algorithm in graph-theoretic terms. We will take $\lambda$-terms to be graphs: a term is a finite DAG with labelled edges and nodes. Every vertex is labelled as either a $\lambda$, an application, or a variable-reference vertex. We will frequently use $l$ for $\lambda$ vertices, $a$ for applications, $x, y$ for variable vertices, and $n$ for general vertices. Edges are labelled with one of $\{\mathrm{body}, \mathrm{fun}, \mathrm{arg}, \mathrm{bvar}\}$. We will frequently use a "dotted accessor" notation, *e.g.*, writing $l.\mathrm{body}$ for the vertex at the end of (the presumed unique) $\mathrm{body}$ edge beginning at node $l$. A $\lambda$ node is connected to its body by a $\mathrm{body}$ edge; and to its bound variable by a $\mathrm{bvar}$ edge. An application node is connected to its function by a $\mathrm{fun}$ edge; and to its argument by an $\mathrm{arg}$ edge. We assume enough

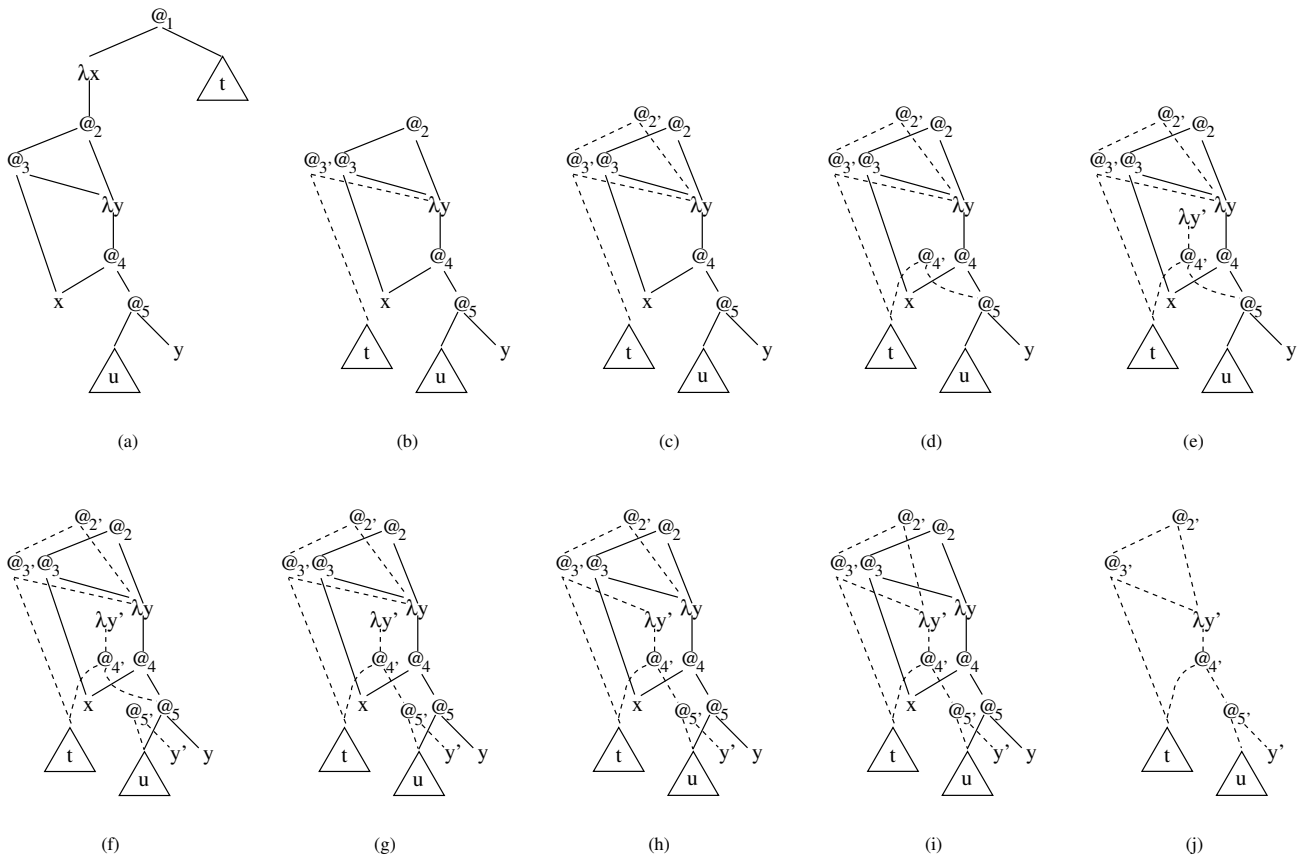



Figure 5: A trace of a bottom-up reduction of the term $(\lambda x.(x(\lambda y.x(uy)))(\lambda y.x(uy)))t$, where the $(\lambda y.x(uy))$ term is shared, and sub-terms $t$ and $u$ are not specified.

constraints on labelling and graph structure to preserve the syntactic structure of the $\lambda$-calculus, *e.g.*, the out-degree of a variable-reference node is zero. We also assume a lexical-scoping resolution of variable reference, *i.e.*, that binding dominates reference: any path from the root to a variable vertex must go through the $\lambda$ vertex binding that variable.

Define $\Lambda_{\mathrm{dag}}$ to be the set of DAGs satisfying these constraints. The classic $\lambda$-calculus is defined in terms of trees; define $\Lambda_{\mathrm{tree}}$ to be the elements of $\Lambda_{\mathrm{dag}}$ that are trees (where we do not count $\mathrm{bvar}$ edges in determining tree-ness of the DAG). The idea behind our DAGs is that they are simply a compact representation for these trees, so we should define the "meaning" of a DAG simply to be the tree to which it "unfolds." We specify this by defining a DAG-to-tree unfolding function, $\mathrm{unfold} : \Lambda_{\mathrm{dag}} \to \Lambda_{\mathrm{tree}}$.

$$\begin{aligned}\mathrm{unfold}\, x &= x \\ \mathrm{unfold}\, a &= (\mathrm{unfold}\, a.\mathrm{fun}) \mathbin{@} (\mathrm{unfold}\, a.\mathrm{arg}) \\ \mathrm{unfold}\, l &= \lambda x\ .\ (\mathrm{unfold}\, l.body) \\ &\quad \text{where } x = l.bvar\end{aligned}$$

(Here, we take "$n_1 \mathbin{@} n_2$" to mean the tree whose root is an application node with function child $n_1$ and argument child $n_2$.) This recursively-defined function is well-defined, as unfold's domain of finite DAGS is well ordered. (Note that we are finessing two details here. First, unfolding a DAG into a tree doesn't bother to replicate variable-reference leaf nodes, which violates "treeness." Second, when replicating a multiply-referenced $\lambda$-expression, unfold doesn't bother to create a unique variable node for each result $\lambda$-expression. If we interpret unfold's result under the standard rules of the $\lambda$-calculus, however, all is well: lexical scope guarantees that the reference/referent connections are invariant across unfolding. Alternately, we could also easily patch the unfold function to allocate fresh variable-reference nodes.)

The unfold function helps makes precise the correctness requirement for our algorithm: reducing a subterm in a DAG, and then unfolding the result, should be the same as unfolding the original DAG and the reducing the corresponding subterms in the tree. Notice that if a redex in a DAG has multiple parents (or, generally, multiple paths to the root), then there will be multiple corresponding redexes in the unfolded tree; reducing this redex in the DAG is equivalent to reducing all of the corresponding redexes in the tree.

In order to connect the state of the graph at intermediate points in the algorithm to the hoped-for final result, we also need to define a "predictive unfold" function:

```
Punfold(g,n,σ) =
  if new?(n) then
    case n of
      x => x
      l => λ x . (Punfold(g,l.body,σ))
           where x = l.bvar
      a => (Punfold(g,a.fun,σ)) @ (Punfold(g,a.arg,σ))
  else σ(unfold(n,g))
```

```
Sub(g,old,new,top,σ) =
    iter λ(par,rel) g . uc(new,par,rel,top,σ,g)
         g
         parlinks(old,g0)

iter f g ∅ = g
iter f g ({x}∪S) = iter f (f x g) S
```

```
uc(new, l, body, top, σ, g) =
  (* Upcopy into λ node l along body edge *)
  old = l.body
  y = l.bvar
  y' = freshnode var
  g += ({y'}, ∅, {(y,y')})
  g = Sub(g, y, y', old, [y↦y']σ, g)
  l' = freshnode λ
  g += ({l'}, {l' -bvar-> y', l' -body-> new}, {(l,l')})
  g = Sub(g, l, l', top, σ)
  return g
```

```
uc(new, a, fun, top, σ, g) =
  (* Upcopy into app node a along fun edge *)
  if (a,a') ∈ g.ρ then
      replace a' -fun-> n edge
         with a' -fun-> new edge
         in g
      return g
  else
      a' = freshnode app
      g += ({a'}, {a' -fun-> new, a' -arg-> a.arg}, {(a,a')})
      g = Sub(g, a, a', top, σ)
      return g

uc(new, a, arg, top, σ, g) =
  (* Upcopy into app node a along arg edge *)
  analogous to fun-edge case
```

Figure 6: The core substitution algorithm, in abstract form.

Punfold takes a graph, a root vertex $n$, and a substitution $\sigma$, which is the DAG-analog of a substitution from the classical $\lambda$-calculus. We model a substitution as a partial function from (variable) nodes to (general) nodes:

$$\sigma \in \text{Subst} = \text{Node} \rightharpoonup \text{Node}$$

(When a substitution is applied in a tree context, we implicitly unfold its result.) Punfold has two "modes," depending on whether it is given *old* graph structure (that is, pre-reduction), or *new* graph structure (that is, created as part of the possibly on-going reduction process). Old structure is simply unfolded and then substituted—that is, we "predict" what the algorithm will do when it gets to that structure. New structure, however, is simply unfolded as-is. We can answer Punfold's old/new question by assuming the original, pre-reduction graph is available; we'll refer to this graph as $g_0$.

With this structure in place, the game plan should now begin to emerge. Punfold$(g_0, b, [x \mapsto n])$ produces the contractum for DAG-redex $(\lambda x.b)n$. Further, the key algorithmic invariant involves Punfold, which will produce the same result throughout the processing of the reduction—the predictive part of Punfold covers parts of the DAG the algorithm has not yet processed. But as the algorithm progresses, Punfold will do less and less prediction, and more and more simple unfolding. By the end of the algorithm, Punfold will do no prediction at all (that is, no element of old structure will lay within the domain of the initial substitution), so that Punfold will be equivalent to unfold on the final result. So, we may conclude that unfolding the final result is what we wanted.

This, coupled with a progress result to guarantee termination on a finite DAG, provides us a correctness claim for the algorithm.

Figure 6 presents the core algorithm as a pair of mathematical functions: Sub, which implements substitution, and uc, which implements the upcopy operation; they are defined in the same sort of "pseudo functional programming language" style we used for Punfold. Note that the auxiliary function, iter, to be well-defined, must remove elements from $S$ in some determined order. We write $\text{parlinks}(old, g_0)$ to mean the set of parent/edge-label pairs describing *old*'s uplinks in the original graph $g_0$. We use the expression "freshnode *label*" to mean the selection of a fresh vertex, labelled *label*, that is not occurring in the implied current graph.

The algorithm operates on graphs $g$ that are represented by a triple $(N, E, \rho)$ of labelled nodes, labelled edges, and a cache $\rho$; the cache stores the mapping from old, copied nodes to their new nodes, as they are created by the upcopy steps. The cache represents the collective behaviour of the cache slot in the algorithm. We model these with finite, partial functions from nodes to nodes:

$$\rho \in \text{Cache} = \text{Node} \rightharpoonup \text{Node}.$$

Where the algorithm stores an item $n'$ in the cache field of some node $n$, the mathematical function adds an $(n, n')$ entry to the current cache $\rho$. To aid the statement of invariants, the abstract algorithm caches the *old* $\mapsto$ *new* translations for *all* copied nodes, not just application nodes (as the actual algorithm does). This is also why substitutions $\sigma$ are passed around the functions—to allow us to use them in our statement of invariants. That is, we need them for the correctness proof. They are not actually needed to compute the result, and so do not appear in an actual implementation of the algorithm. We use dotted-accessor notation, $g.\rho$, to write the cache of a given graph $g$.

The key internal invariant of the algorithm is that entries in the cache are related by Punfold—if we see an $(n, n')$ entry in the cache, then $n'$ represents a substitution that has been, or is in the midst of being performed on, $n$. Because the substitution may not be complete, we must unfold $n'$ with the *predictive* unfolder, which will interpolate the uncompleted parts of the substitution into the result tree for us. For convenience, we will name this main invariant "CG," for "cache good:"

$$\text{CG}[\sigma, g] \;\equiv\; \forall (n, n') \in g.\rho \;.\; \text{Punfold}(g, n', \sigma) \stackrel{\alpha}{=} \sigma\ \text{unfold}(g, n).$$

The reason this relation needs a general substitution is because of the extra copying cascades triggered when the algorithm copies up through a $\lambda$-expression, as described in section 5. The extra $y \mapsto y'$ variable substitutions we must perform get lumped into the substitution underneath the copied $\lambda$-expression (but are not needed *above* the copied $\lambda$-expression). Again, the substitutions are passed around the functions primarily as bookkeeping device that allows us to assert the necessary invariants at different points in the function definition.

The main "entry point" for the reduction algorithm is the $\text{Sub}(g, old, new, top, \sigma)$ function, which performs an $[old \mapsto new]$ substitution in $g$, returning the augmented graph. The preconditions

for calling Sub are

$\exists top' \,.\, (top, top') \in g.\rho$
*top* dominates *old* in $g_0$
CG$[\sigma, g]$
$(old, new) \in g.\rho$
*old* not reachable from *new* in $g$
$g_0 \subset g$

The *top* parameter must be some node that dominates *old* in the original graph $g_0$. We also require it to be in the cache; thus *top* acts as a terminating "sentinel." We also require the $(old, new)$ pair to be in the cache.

Consider the initial call to Sub, in response to a desire to reduce some redex $(\lambda x.n)m$. Assume that *top* is the topmost application under $n$—if there is no such node, then simple special-cases apply, as discussed in section 5. Let

$$\begin{aligned} top' &= \text{a fresh node not in } g_0, \\ \rho' &= \rho \cup \{(x, m), (top, top')\}, \text{ and} \\ e' &= \{top' \xrightarrow{\text{fun}} top.\text{fun},\ top' \xrightarrow{\text{arg}} top.\text{arg}\} \\ g_1 &= g_0 \cup (\{top'\}, e', \rho'). \end{aligned}$$

Then Sub$(g_1, x, m, top, [x \mapsto m])$ satisfies the preconditions.

If we let $g'$ be Sub's returned value, then its post-conditions are

$g_0 \subset g'$
$g.\rho \subset g'.\rho$
CG$[\sigma, g']$
*old* not reachable from *top*$'$ in $g'$.

When applied to the initial call, this is enough to give us

$$\begin{aligned} [x \mapsto m]\ \text{unfold}(g_0, top) &\stackrel{\alpha}{=} \text{Punfold}(g', top', [x \mapsto m]) \\ &= \text{unfold}(g', top'), \end{aligned}$$

which is precisely what we want of our substitution.

Sub is a fairly simple function; it just uses the uc function to initiate *old*$\mapsto$*new* upcopies through every link from *old* up to some parent. The presence of *top* in the cache causes all of these upcopies to terminate; the new graph structure they create is entered into the cache. The pre-conditions for uc$(new, par, rel, top, \sigma, g)$ are

$(par.rel, new) \in g.\rho$
$\exists top' \,.\, (top, top') \in g.\rho$
CG$[\sigma, g]$
*par.rel* not reachable from *new* in $g$
*top* dominates *par* in $g_0$
$g_0 \subset g$

and, if we let $g'$ be the returned value, its post-conditions are

$g.\rho \subset g'.\rho$
CG$[\sigma, g']$
*par* is not reachable from *top*$'$ in $g'$.
$g_0 \subset g'$

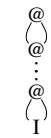
The uc function is defined in figure 6 in three cases: upcopying into a $\lambda$-expression, upcopying through a fun link into an application node, and upcopying through an arg link into an application node (this last case is not given; it is entirely similar to the arg-link case). Consider, first, the $\lambda$ case. Note how the pre-Sub code sets up the preconditions for the first Sub call, which triggers the $y \mapsto y'$ variable-substitution cascade—in particular, note the $[y \mapsto y']$ mapping must be added to the prediction substitution $\sigma$. This step sets up the pre-conditions for the upward continuation of the algorithm, which substitutes $\lambda y'.new$ for $\lambda y.old$, completing the upcopy.

## 9 Experiments

To gain experience with the algorithm, a pair of Georgia Tech undergraduates implemented three $\beta$-reduction algorithms in SML: the bottom-up algorithm, a reducer based on the suspension $\lambda$-calculus (see section 10.1), and a simple, base-line reducer, based on the simple top-down, blind-search `subst` procedure shown in section 1. They also built a tool-chain that allowed us to convert between a concrete s-expression form for terms and the particular representations required for the various algorithms. This allowed us to test the algorithms by running all three on randomly constructed trees, and comparing the results for discrepancies.

We then implemented two normalisers for each algorithm, one evaluating to normal form; the other, to weak-head normal form. The normalisers are called with an integer "reduction budget"—after performing that many reductions, they give up.

We first ran a few tests designed to show the bottom-up algorithm at its best and worst. The first test reduced a "chain of pearls" stack of applications, 20 deep, with full sharing and the identity combinator $I = \lambda x.x$ at the leaf:




This set up the bottom-up algorithm to do very well, since it can exploit the sharing to achieve exponential speedup. (Note, in particular, that the bottom-up algorithm is what the graph-reduction community calls "fully lazy." That is, reductions on dynamically-created shared structure are seen by all parents of that structure.)

Second, we normalised a 40,000-long chain of the form $(\lambda y.\lambda x.\lambda x.\lambda x \ldots \lambda x.y)\ \lambda z.z$. This is an example where even the naïve algorithm can do well, as there is no search involved—the branch factor of a chain of nested $\lambda$-expressions is one. So this shows off the "constant factors" of the algorithm.

Third, we normalised a full binary tree, 20 deep, of applications, with identity combinators at the leaves. This is the same term as the one in the first test, but with no sharing of structure. This is designed to show the algorithm at its worst, since there is no sharing at all during the entire normalisation process.

All normalisations were executed with a budget of 1,000 reductions. Here are the timings from these runs:

| | 20pearls | 40k$\lambda$ | tree20 |
|---|---|---|---|
| BU | 0+0 | 15+0 | 9+84 |
| BU-nocheck | 0+0 | 318+1433 | 10+80 |
| SLC | N/A | 70+73 | 16+72 |
| Simple | N/A | 364+1536 | 1+3 |

The "BU-nocheck" entries are for the bottom-up algorithm with the fast-path single-parent optimisation turned off. The tests were run using SML/NJ 110.42 on a 667Mhz G4 Powerbook with 512MB of PC133 RAM under osX 10.2.2. The 20-length chain-of-pearls entry could not be run for SLC or the simple algorithm, as it requires a representation that can handle DAGs; the numbers for these algorithms are essentially those for the equivalent, unfolded 20-deep full application tree in the third column. Each pair of numbers $e+g$ is the time in milliseconds $e$ to execute the algorithm, and the time $g$ spent in garbage collection. Note that the bottom-up algorithm is able to polish off the 40,000-element chain of nested $\lambda$-expressions easily, as this can be handled by the single-parent fast path in constant time.

Experience with these runs, and others, led us to implement a tightly coded C implementation of the same three algorithms to get

more accurate measurements. The SML version had several issues that affected measurement. The implementation contains some redundant safety tests that the type system is unable to eliminate. SML also limits our ability to do detailed layout of the data structures. For example, by allocating the uplink structure in the same block of storage as the parent node to which it refers, we can move from the uplink to the referenced parent node with pointer arithmetic (*i.e.*, on-processor), instead of needing to do a slower memory load. Similarly, if we allocate a $\lambda$-expression and its bound variable together, in a single memory block, we can move between these two structures without doing memory traffic. Finally, C eliminated our dependence on the garbage collector. The bottom-up algorithm intrinsically does its own storage management, so we don't need a GC. We get a fair amount of GC overhead in our SML implementation, SML/NJ, as it allocates even procedure call frames in the heap.

The SLC and simple reducers managed storage in the C implementation with the Boehm-Demers-Weiser garbage collector, version 6.2. (Perhaps it is possible to explicitly manage storage in the suspension $\lambda$-calculus as well, but we did not see a simple way to do this.)

We compiled the code with gcc 2.95.4 -g -O2 -Wall and performed the test runs on an 800 MHz PIII (256 KB cache), 128 MB RAM, Debian GNU/Linux 3.0 system. Here are the timings for these runs:

| | CPU time (ms) | | | # reductions | |
|---|---|---|---|---|---|
| | BUBS | SLC | Simple | BUBS | Tree |
| (fact 2) | 0 | 10 | 10 | 123 | 180 |
| (fact 3) | 0 | 20 | 20 | 188 | 388 |
| (fact 4) | 0 | 40 | $\infty$ | 286 | 827 |
| (fact 5) | 0 | 160 | $\infty$ | 509 | 2045 |
| (fact 6) | 10 | 860 | $\infty$ | 1439 | 7082 |
| (fact 7) | 20 | 5620 | $\infty$ | 7300 | 36180 |
| (fact 8) | 190 | 48600 | $\infty$ | 52772 | 245469 |
| nasty-I | 30 | 740 | $\infty$ | 7300 | 8664 |
| pearl10 | 0 | N/A | N/A | 10 | N/A |
| pearl18 | 0 | N/A | N/A | 18 | N/A |
| tree10 | 0 | 0 | 0 | 1023 | 1023 |
| tree18 | 740 | 2530 | 1980 | 262143 | 262143 |

The system gave us a measurement precision of 10 ms; an entry of 0ms means below the resolution of the timer—*i.e.*, less than 10ms; a measurement of $\infty$ means the measurement was halted at 10 cpu-minutes (or, in one case, at 2min 20sec due to severe page thrashing).

Fact entries are factorial terms, with Church-numeral encodings. Nasty-I is a complex, hand-generated, tree of K and S combinators that reduces to I; the tree contains 20,152 nodes. The "pearl$i$" and "tree$i$" terms are as described for the SML timings.

We caution the reader from drawing too much from these timings. They are fairly preliminary. We are undertaking to perform a large series of random tests, and generate performance statistics, in order to get a better understanding of the algorithm's performance; we hope to report on this in the final version of the paper.

We also do not wish to claim that the bottom-up algorithm is a competitive stand-alone graph-reduction system. Modern graph-reducers are highly-engineered systems that employ a battery of static analyses and run-time optimisations to gain performance. The bottom-up reducer, in contrast, embodies a single idea. Perhaps this idea could be applied to the production of a competitive graph-reducer; we chose term normalisation simply as a generic task that would thoroughly exercise the reduction engines. We are also quite willing to believe that a complex algorithm such as the read phase of the suspension $\lambda$-calculus has opportunities for clever speedups that our simple implementation did not implement. Further, the experiments we performed do not show the SLC algorithm at its best—it is specifically tuned to provide an extra capability that, for applications such as theorem provers, can provide tremendous "application-level" speedups: laziness. Normalisation examines the entire tree, thus eliminating some of the benefit of SLC's laziness.

That all said, the bottom-up algorithm is obviously very fast. Part of this speed comes from "full laziness:" a reduction of a shared redex is shared by the parents of the redex. But this is not the whole story—the (fact 8) case does 1/5 of the reductions, but gets a speedup of 256x, and the tree18 case has no sharing at all, but still manages a speedup of 3.4x (over SLC, that is; the speedups over the simple reducer are different, but the general picture is similar). This is primarily due to the elimination of blind search, and consequent ability to share structure *across* a reduction step (as opposed to *within* a term).

One of the striking characteristics of the bottom-up algorithm is not only how fast it is, but how well-behaved it seems to be. The other algorithms we've tried have fast cases, but also other cases that cause them to blow up fairly badly. The bottom-up algorithm reliably turns in good numbers. We conjecture this is the benefit of the being able to exploit both sharing and non-sharing as they arise in the DAG. If there's sharing, we benefit from re-using work. If there's no sharing, we can exploit the single-parent fast path. These complementary techniques may combine to prevent the algorithm from being susceptible to particular inputs.

## 10 Related work

A tremendous amount of prior work has been carried out exploring different ways to implement $\beta$-reduction efficiently. In large part this is due to $\beta$-reduction lying at the heart of the graph-reduction engines that are used to execute lazy functional languages. The text by Peyton Jones *et al.* [10] summarises this whole area very well.

However, the focus of the lazy-language community is on representations tuned for *execution*, and the technology they have developed is cleverly specialised to serve this need. This means, for example, that it's fair game to fix on a particular reduction order. For example, graph reducers that overwrite nodes rely on their normalisation order to keep the necessary indirection nodes from stacking up pathologically. A compiler, in contrast, is a $\lambda$-calculus client that makes reductions in a less predictable order, as analyses reveal opportunities for transformation.

Also, an implementation tuned for execution has license to encode terms, or parts of terms, in a form not available for examination, but, rather, purely for execution. This is precisely what the technique of supercombinator compilation does. Our primary interest at the beginning of this whole effort was instead to work in a setting where the term being reduced is always directly available for examination—again, serving the needs of a compiler, which wants to manipulate and examine terms, not execute them.

### 10.1 The suspension $\lambda$-calculus

Nadathur *et al.* [9] have developed an extremely clever representation for $\lambda$-calculus terms called the "suspension $\lambda$-calculus" (SLC), that extends the language to allow environments and, hence, closures over them, to be represented syntactically. This means $\beta$-reduction can be done in constant time, by simply producing a term that syntactically encodes a closure over the $\lambda$-expression's body in an environment mapping its variable to the redex's argument term. Additionally, the language requires terms to be converted to a "nameless" de Bruijn-index representation.

The point of doing so is laziness, of two particular forms. First, we can reduce a closure term incrementally, doing only enough work to resolve whether its top-level is a variable reference, an application or a $\lambda$-expression, while leaving any child terms (such as the body of a $\lambda$-expression) suspended, *i.e.*, explicit closures. Thus the work of completely resolving a closure created by a reduction into a tree composed of simple variable/application/$\lambda$-expression terms can be done on a pay-as-you-go basis. For example, a theorem prover that wishes to compare two terms for equality can recursively explore the terms, resolving on the fly, but abandon the recursive comparison as soon as two subterms fail to match. By lazily resolving the tree, no work is done to resolve parts that are not needed.

Second, the reduction system for the expanded language includes rules for merging two substitution environments together before applying the single compound substitution to a term. This converts a double pass over the term into a single pass.

The great payoff for using a term-manipulation engine based on the SLC comes for systems that can exploit laziness. A program that examines the entire tree produced by a given reduction, however, is not going to benefit as much from the laziness. A compiler, for example, is a program that typically "walks" the entire program structure given to it, performing analyses and transforms on the structure.

SLC reduction, in the terms we've defined, uses "blind search" to find the variables being substituted. This cost is mitigated by its ability to merge environments—it increases the odds that searching down a particular link will turn up some variable that needs to be replaced. On the other hand, this is, to some degree, just shifting work back to the environment-merging calculations, which are not trivial.

The SLC also has strong barriers to sharing internal structure as a DAG. This is a consequence of its representation of terms using de Bruijn indices—a term with free variable references will have two distinct forms at two different places in the tree. Again, this is somewhat mitigated by the SLC's ability to place a term in an environment and then produce a simple closure (which might contain multiple references to the bound term) with that environment. However, by the time a final, base term has been completely produced, all environments must be removed, and so the replication has to be performed at some point.

The uplinked $\lambda$-DAG representation explicitly provides for sharing, and does so in a way that allows clients of the data structure to operate on the entire structure *without having to unfold it into its equivalent tree*, an operation that could induce exponential blowup in space.

Finally, the SLC is a quite sophisticated algorithm. The fine details of its reduction rules are fairly subtle and non-obvious. It requires transforming the term structure into a related, but very distinct form: nameless terms, with new and complex non-terminals expressing suspended reductions and their environments, in various states of merging.

The charm of the bottom-up technique presented here is its simplicity. The data structure is essentially just a simple description of the basic syntax as a datatype, with the single addition of child→parent backpointers. It would generalise easily to the richer languages used by real compilers and other language-manipulation systems. It's very simple to examine this data structure during processing; very easy to debug the reduction engine itself. There are really only two important invariants on the structure: (1) all variables are in scope (any path upwards from a variable reference to the root must go through the variable's binding $\lambda$-expression), and (2) uplink backpointers mirror downlink references.

### 10.2 Director trees

Director trees [4] are a representation driven by the same core issue that motivates our uplink-DAG representation: they provide a way to guide search when performing $\beta$-reduction. In the case of director trees, however, we can do our search top-down.

At each application node, we sort the free variables of that term by lexical height. Then, we annotate each application node with a string of symbols drawn from the set $\{/, \setminus, \wedge, 0\}$, one symbol for each free variable. The symbol used for a given variable tells if the variable occurs in the left child only, the right child only, both children, or neither child, respectively. (The 0 symbol is only used in degenerate cases.) These strings provides the information we need to do top-down guided search; with this information in hand, we can do $\beta$-reduction more or less just as we did insertion into an ordered binary tree in section 2.

Director trees, however, can impose a quadratic space penalty on our trees. The standard example showing this is the term $\lambda x_1 \ldots \lambda x_n.(x_n \ldots x_1)$. Uplinked $\lambda$-DAGs are guaranteed to have linear space requirements. Whether or not the space requirements for a director-tree representation will blow up in practice depends, of course, on the terms being manipulated. But the attraction of a linear-space representation is knowing that blow-up is completely impossible.

Like the suspension $\lambda$-calculus, director trees also have the disadvantage of not being a direct representation of the original term; there is some translation involved in converting a $\lambda$-calculus term into a director tree.

### 10.3 Optimal $\lambda$ reduction

The theory of "optimal $\lambda$ reduction" [7, 6, 3] (or, OLR), originated by Lévy and Lamping, and developed by Abadi, Asperti, Gonthier, Lawall, Mairson *et al.*, is a body of work that shares much with bottom-up $\beta$-reduction. Both represent $\lambda$-terms using graph structure, and the key idea of connecting variable-binders directly to value-consumers of the bound variable is present in both frameworks—and for the same reason, namely, from a desire that substitution should be proportional to the number of references to the bound variable, removing the need to blindly search a term looking for these references.

However, the two systems are quite different in their details, in fairly deep ways. The "Lamping graphs" of optimal $\lambda$ reduction add extra structure to the graph, in the form of "croissants" and "brackets," to allow a novel capability: *incremental* $\beta$-reduction. Reduction is not an atomic operation in OLR. It can be performed in multiple steps; intermediate stages of the reduction are valid graphs. (The croissant and bracket marks delimit the boundaries of a reduction step as they propagate through the graph.) This is an enormous difference with our simple bottom-up $\beta$-reduction system—it is, essentially, an exciting and different model of computation from the one based on the classical $\lambda$-calculus. However, it comes with a cost: the greatly increased complexity of the graph structure and its associated operations. The details of this complexity have prevented OLR-based systems from being used in practice. In particular, in actual use, the croissant and bracket marks can frequently pile up uselessly along an edge, tying up storage and processing steps. It also makes it difficult to "read" information from the graph structure. In this respect, OLR remains the province of theoreticians, not implementors.

Finally, optimal $\lambda$ reduction comes with a great deal of theoretical underpinnings. Of particular note is that it makes a claim to optimality, in terms of using sharing to guarantee the minimal number of reductions. We make no such claim. It is clear that the bottom-up algorithm exploits sharing to reduce work, and the

experimental timings given in the previous section show just how much speed is gained over other implementations. But is it optimal? We have no idea.

## 11 Future work

We have several topics we would like to pursue, related to the use of uplinked $\lambda$-DAGs.

**Other operations** $\beta$-reduction is, of course, not the only operation one might wish to perform on terms of the $\lambda$-calculus. In a DAG representation, we also might wish to provide a parent-splitting operation, to unshare a node that has multiple parents. This means cloning the child node, and dividing the parents between the original child and its copy as indicated by some partition. This operation, which is easy to implement, would be useful for a compiler that made reduction (aka "inlining") decisions based on context. If we want to contract a redex in one context, but leave it as-is in another, we must first replicate the node, so the two contexts have distinct redexes. Note that only the top of the redex needs to be replicated; the two redex copies can share children, so this operation is fast. (Cloning a $\lambda$-expression is more work than cloning an application, however—the $\lambda$'s bound variable must also be replicated to preserve $\alpha$-uniqueness; this kicks off a cascade of up-copying along the paths from the variable up through the $\lambda$-expression being cloned, just as in the $\beta$-reduction case.)

Comparing terms for equality brings up the question of which definition of equality we mean. The spectrum runs from complete textual equality to tree equality modulo $\alpha$-conversion to extensional equality in the model of a denotational semantics. The presence of sharing in our DAG representation raises new distinctions, as well. One useful definition of equality for DAGs is: if we expanded term DAGs $t_1$ and $t_2$ into their equivalent trees, would these trees be structurally equal, that is, equal modulo $\alpha$-conversion? Implementing this efficiently is a nice puzzle. Note that we *can't* use the usual trick of converting to de Bruijn indices and comparing the results—the DAG representation completely rules out the use of de Bruijn indices, as there may be two paths from a variable up to its binding $\lambda$-expression that run through different numbers of intermediate $\lambda$-expressions!

We have designed and implemented an algorithm for this equality test that is "almost linear" (in the sense of the inverse Ackermann function) in the sizes of the DAGs when the answer is "yes;" the algorithm uses the fast amortised union-find algorithm for its speed. A detailed discussion of the $\alpha$-DAG equality algorithm is beyond the scope of the article; we expect to describe it fully in a longer report.

We would also like to consider the design of an efficient hash function that is insensitive to sharing or $\alpha$-conversion, *i.e.*, one that respects the equality test outlined above.

**Variants of the algorithm** It would be interesting to see if the algorithm could be adapted to operate on general graph structure, as opposed to DAGs. This would permit recursion to be captured with circular structure, as opposed to encoding it using syntactic devices such as the Y combinator.

The basic marking-based search&copy technique is one that works on general graphs with no trouble. However, one complication in such a framework is that a reduction that unrolls a recursion can cause uplinks to become downlinks. This perturbs some of the fundamental invariants on which the algorithm is based, and so affects much of the code. Altering the algorithm to properly account for this behavior would require careful thought.

**Integrating with orthogonal implementation techniques** As we've noted already, there is tremendous body of work on the high-performance graph reduction of $\lambda$-calculus terms. The uplinked $\lambda$-DAG representation is not mutually exclusive with many of these techniques. It would be worth investigating to discover how much of the "classical" graph-reduction technology could be applied in this DAG framework. For example, could supercombinators be compiled into native code to operate on uplinked $\lambda$-DAG representations?

**DAG-based compiler** We are very interested in trying to put the bottom-up representation to use in a real compiler, to represent both program terms and sophisticated types. Type-based compilers [12, 11], in particular, are notorious for term explosion in the intermediate type terms; the sharing introduced by the bottom-up algorithm has potential to help here.

A compiler is an application that typically produces output proportional to the size of the intermediate code tree, which, in our case, is really the *unfolded* intermediate code tree. So perhaps there is less sharing payoff in representing program terms with a $\lambda$-DAG (as opposed to the type terms). However, this is only true of the *final* program term—program transforms and analyses could well benefit from sharing-based compression of the program term, avoiding term explosion in the intermediate stages. Further, even if the target language (*i.e.*, assembler) doesn't allow sharing, we can still benefit from generating code from a DAG by caching the program-term-to-assembler translations. Finally, as we've described earlier, the bottom-up reduction algorithm gets time and space savings not only from sharing *within* a $\lambda$-term, but also from the elimination of blind search and the associated sharing of structure *across* a reduction—*i.e.*, the reduction algorithm tries to copy as little graph structure as possible when reducing.

Compilers typically operate on a tree-based intermediate representation. It almost seems that humans naturally gravitate to tree computations in preference to graph or DAG computations, as the former can be done with simple recursive traversals, frequently in a pure functional style; while the latter require side-effecting search&mark algorithms with subtler fixed-point properties.

It is intriguing, nonetheless, to consider what a compiler would be like that was more fundamentally based on representing program structure as a DAG (as opposed to using the bottom-up representation essentially as a short-hand for a tree). Costs and benefits are not always what they seem in this setting. For example, inlining a procedure definition by replacing the procedure's name with its $\lambda$-expression has no space cost in a DAG representation, even if the procedure is invoked at multiple call sites. Specialising such an application by subsequently contracting the $\lambda$-expression's application to a particular set of arguments is what causes code replication—although, even in this case, the bottom-up reduction algorithm attempts to share common structure.

Even more intriguing and exotic is the possibility of allowing general graph structure for our DAG. The question of variable scope (a tree notion), for example, becomes the more general question of binders dominating references (a graph-theoretic notion). We begin to verge, at this point, from the realm of $\lambda$-calculus to the realm of "flat" SSA representations [5]. The challenge is to do so, and yet retain the *ideas*—in their suitably generalised form—of scope and closure and higher-order functional values from the $\lambda$-calculus setting. At least one such compiler has been written, by Bawden [2], though the effort was never written up and published.

We do not claim that some sort of general-graph/circular-structure variant of the $\lambda$-calculus is a better way to build compilers. We do think it is an interesting idea to consider.

## 12 Conclusion

We've introduced a pair of simple core ideas:

- representing $\lambda$-calculus terms as DAGs, to permit sharing introduced naturally by $\beta$-reduction; and
- introducing child→parent backpointers and $\lambda$→variable links to efficiently direct search and construction.

We certainly are not the first to consider using DAGs to represent terms of the $\lambda$-calculus; the key point we are making is that these two ideas work together. In order to exploit the backpointers, we need the DAG representation to allow us to build terms without having to replicate the subterm being substituted for the variable. This is the source of speed and space efficiency.

The algorithm is simple and directly represents the term without any obscuring transform, such as combinators, de Bruijn indices or suspensions, a pleasant feature for $\lambda$-calculus clients who need to examine the terms. It is also, in the parlance of the graph-reduction community, fully lazy.

Finally, we'd like to make the point that the data structure embodies a nice "philosophy" of $\lambda$-calculus terms. A variable is nothing but a pointer, from reference to definition. The fundamental point of variables is to allow sharing—we can bind a variable once, and refer to it multiple times. This is a DAG-like concept, but to represent it in a concrete grammar, we introduce names to encode the links syntactically. Once a tool has read in a term, however, it makes sense to boil that term down to its essence. Names are no longer important; only structure is.

Further, once we've taken the step of representing the sharing represented by names directly as a DAG, and do away with names, we might as well extend the privilege of sharing to the rest of the term structure. And so we are led to general DAGs.

In the bottom-up DAG representation, names, once they have served their purpose of communicating the connections, vanish. What we are left with, what our tools have to reason with, is nothing but pure structure.

## 13 Acknowledgements

Bryan Kennedy and Stephen Strickland, undergraduates at Georgia Tech, did the entire implementation and evaluation reported in section 9. Zhong Shao provided helpful discussions on the suspension $\lambda$-calculus. Chris Okasaki and Simon Peyton Jones tutored us on director trees. Harry Mairson and Alan Bawden provided lengthy and patient instruction on the subtleties of optimal lambda reduction and Lamping graphs.

## References

[1] Henk Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984.

[2] Alan Bawden. Personal communication, November 2002. Alan wrote the compiler, a toy exercise for Scheme, sometime in the late 1980's.

[3] Georges Gonthier, Martín Abadi and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, January 1992.

[4] J. R. Kennaway and M. R. Sleep. Director strings as combinators. ACM Transactions on Programming Languages and Systems, 10, pages 602–626, (October 1988).

[5] Richard A. Kelsey. A correspondence between continuation-passing style and static single assignment form. In *ACM SIGPLAN Workshop on Intermediate Representations*, *SIGPLAN Notices*, vol. 30, no. 3, pages 13–22, January 1995.

[6] John Lamping. An algorithm for optimal lambda-calculus reduction. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30, January 1990.

[7] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda-calcul.* Ph.D. thesis, Université Paris VII, 1978.

[8] R. Milner, M. Tofte, R. Harper, D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[9] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science* 198(1–2):49–98, May 1998.

[10] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[11] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, *SIGPLAN Notices 30(6)*, pages 116–129, June 1995.

[12] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, *SIGPLAN Notices 31(5)*, pages 181–192, May 1996.

## A BetaSub.sml

Not including comments, blank lines, or a simple doubly-linked list library, the source code for the core data structures and $\beta$-reduction algorithm is 180 lines of SML code. Here is a complete listing, copiously commented. This code listing is supplementary material to the main body of the paper; if the program committee feels this violates the spirit of the page limitations on submissions, feel free to omit these following pages. It is included at least to give you the choice.

```
(* Bottom-up Beta Substitution *)

structure DL = DoubleLists

(* Core datatype definitions
 ***************************************************************************
 * There are three kinds of nodes: lambdas, var refs and applications.
 * Each kind gets its own ML datatype, instead of having a single,
 * three-constructor datatype. Why? It allows us to encode more structure
 * in the ML type system. E.g., the *parent* of a node can only be a lambda
 * or an app; not a var-ref. So we can define a two-constructor node-parent
 * type, ruling out the var-ref possibility. And so forth.
 *
 * Note, also, that some of these "foo option ref" record fields are because we
 * are constructing circular structure. Backpointers are initialised to
 * "ref NONE," then we slam in "SOME <node>" after we have later created <node>.
 *)

(* bodyRef is the parent record belonging to our child node (our body) that
 * points back to us. I.e., suppose our body node N has three parents, of
 * which we are one. Then N has a three-element doubly-linked list (DLL)
 * of parent records, one for each parent. The one that points back to us
 * is the record sitting in *our* "bodyRef" field. This allows us to delink
 * ourselves from the child's parent list & detach the child in constant time
 * when copying up through the lambda node.
 *)
datatype LambdaType = Lambda of {var: VarType, body: Term option ref,
                                 bodyRef: ChildCell DL.dl option ref,
                                 parents: ChildCell DL.dl ref,
                                 uniq: int}


(* funcRef and argRef are similar to the bodyRef field
 * of the LambdaType record above.
 *)
and AppType = App of {func: Term option ref, arg: Term option ref,
                      funcRef : ChildCell DL.dl option ref,
                      argRef  : ChildCell DL.dl option ref,
                      copy: AppType option ref,
                      parents: ChildCell DL.dl ref,
                      uniq:int}

and VarType = Var of {name: string,
                      parents: ChildCell DL.dl ref,
                      uniq:int}

and Term = LambdaT of LambdaType        (* Type of a general LC node. *)
         | AppT of AppType
         | VarT of VarType

(* This tells us what our relationship to our parents is. *)
and ChildCell = AppFunc of AppType
              | AppArg  of AppType
              | LambdaBody of LambdaType

(* Get the parents of a Term. *)
fun termParRef(LambdaT(Lambda{parents, ...})) = parents
```

```
  | termParRef(AppT(App{parents, ...}))       = parents
  | termParRef(VarT(Var{parents, ...}))       = parents


(* A rather subtle point:
*************************************************************************
* When we do upsearch/copying, we chase uplinks/backpointers, copying old tree
* structure, creating new tree structure as we go. But we don't want to search
* up through *new* structure by accident -- that might induce an infinite
* search/copy. Now, the the only way we can have a link from an old node up to
* a new parent is by cloning an app node -- when we create a new app, it has
* one new child NC and one old child OC. So our new app node will be added to
* the parent list of the old child -- and if we should later copy up through
* the old child, OC, we'd copy up through the new app node -- that is, we'd
* copy the copy. This could get us into an infinite loop. (Consider reducing
*   (\x. x x) y
* for example. Infinite-loop city.)
*
* We eliminate this problem in the following way: we don't install *up* links
* to app nodes when we copy. We just make the downlinks from the new app node
* to its two children. So the upcopy search won't ever chase links from old
* structure up to new structure; it will only see old structure.
*
* We *do* install uplinks from a lambda's body to a newly created lambda node,
* but this link always goes from new structure up to new structure, so it will
* never affect the our search through old structure. The only way we can have a
* new parent with an old child is when the parent is an app node.
*
* When we are done, we then make a pass over the new structure, installing the
* func->app-node or arg->app-node uplinks. We do this in the copy-clearing
* pass -- as we wander the old app nodes, clearing their cache slots, we take
* the corresponding new app node and install backpointers from its children
* up to it.
*
* In other words, for app nodes, we only create downlinks, and later bring the
* backpointer uplinks into sync with them.
*)

(* Given a term and a ChildCell, add the childcell to term's parents. *)
fun addToParents(node, cclink) = let val p = termParRef node
                                 in p := DL.add_before(!p, cclink)
                                 end

(* Is dll exactly one elt in length? *)
(* ML pattern matching rules.        *)
fun len1 (DL.Node(_,_,ref DL.NIL)) = true
  | len1 _                         = false


(* clearCopies(redlam, topapp)
 *************************************************************************
 * When we're finished constructing the contractum, we must clean out the
 * app nodes' copy slots (reset them to NONE) to reset everything for the next
 * reduction.
 * - REDLAM is the lambda we reduced.
 *
 * - TOPAPP is the highest app node under the reduced lambda -- it holds
 *   the highest copy slot we have to clear out. If we clear it first, then
 *   we are guaranteed that any upwards copy-clearing search started below it
 *   will terminate upon finding an app w/an empty copy slot.
 *
 * Every lambda from REDLAM down to TOPAPP had its var as the origin of an
 * upcopy:
 * - For REDLAM, the upcopy mapped its var to the redex's argument term.
 * - The other, intermediate lambdas *between* REDLAM & TOPAPP (might be zero
```

```
 *   of these) were copied to fresh lambdas, so their vars were mapped to
 *   fresh vars, too.
 * So, now, for each lambda, we must search upwards from the lambda's var,
 * clearing cached copies at app nodes, stopping when we run into an
 * already-cleared app node.
 *
 * This cache-clearing upsearch is performed by the internal proc cleanUp.
 * (Get it?)
 *
 * When we created fresh app nodes during the upcopy phase, we *didn't*
 * install uplinks from their children up to the app nodes -- this ensures
 * the upcopy doesn't copy copies. So we do it now.
 *)

fun clearCopies(redlam, topapp) =
  let val App{copy=topcopy,...} = topapp                    (* Clear out top*)
      val ref(SOME(App{arg,argRef, func, funcRef,...})) = topcopy
      val _ = topcopy := NONE                               (* app & install*)
      val _ = addToParents(valOf(!arg),  valOf(!argRef));   (* uplinks to   *)
      val _ = addToParents(valOf(!func), valOf(!funcRef));  (* its copy.    *)

      fun cleanUp(AppFunc(App{copy=ref NONE,...})) = ()

         | cleanUp(AppFunc(App{copy as ref(SOME(App{arg, argRef,
                                                func, funcRef,...})),
                               parents,...})) =
           (copy := NONE;
            addToParents(valOf(!arg),  valOf(!argRef));  (* Add uplinks *)
            addToParents(valOf(!func), valOf(!funcRef)); (* to copy.    *)
            DL.app cleanUp (!parents))

         | cleanUp(AppArg(App{copy=ref NONE,...})) = ()

         | cleanUp(AppArg(App{copy as ref(SOME(App{arg, argRef,
                                                  func, funcRef,...})),
                              parents,...})) =
           (copy := NONE;
            addToParents(valOf(!arg),  valOf(!argRef));  (* Add uplinks *)
            addToParents(valOf(!func), valOf(!funcRef)); (* to copy.    *)
            DL.app cleanUp (!parents))

         | cleanUp(LambdaBody(Lambda{parents,var,...})) =
           (varClean var; DL.app cleanUp (!parents))

      and varClean(Var{parents=varpars,...}) = DL.app cleanUp (!varpars)

      fun lambdascan(Lambda{var, body=ref(SOME b),...}) =
              (varClean var;
               case b of LambdaT l => lambdascan l | _ => ())

  in lambdascan redlam
  end

(* freeDeadNode term -> unit
 ***************************************************************************
 * Precondition: (termParents term) is empty -- term has no parents.
 *
 * A node with no parents can be freed. Furthermore, freeing a node
 * means we can remove it from the parent list of its children... and
 * should such a child thus become parentless, it, too can be freed.
 * So we have a recursive/DAG-walking/ref-counting sort of GC algo here.
 *
 * IMPORTANT: In this SML implementation, we don't actually *do* anything
 * with the freed nodes -- we don't, for instance, put them onto a free
 * list for later re-allocation. We just drop them on the floor and let
```

```
 * SML's GC collect them. But it doesn't matter -- this GC algo is *not
 * optional*. We *must* (recursively) delink dead nodes. Why? Because
 * we don't want subsequent up-copies to spend time copying up into dead
 * node subtrees. So we remove them as soon as a beta-reduction makes
 * them dead.
 *
 * So this procedure keeps the upwards back-pointer picture consistent with
 * the "ground truth" down-pointer picture.
 *)
fun freeDeadNode node =
  let
      fun free(AppT(App{func=ref(SOME functerm), funcRef,
                        arg=ref(SOME argterm),   argRef,
                        parents, ...})) =
          (delPar(functerm, valOf(!funcRef));  (* Node no longer parent      *)
           delPar(argterm,  valOf(!argRef)))   (* of func or arg children.   *)

        | free(LambdaT(Lambda{body=ref(SOME bodyterm),   (* Lambda no longer *)
                              bodyRef, parents, ...})) = (* parent of body.  *)
           delPar(bodyterm, valOf(!bodyRef))

        (* We wouldn't actually want to dealloc a parentless var node, because
         * its binding lambda still retains a ref to it. Responsibility for
         * freeing a var node should be given to the code (just above) that
         * freed its lambda.
         *)
        | free(VarT _) = ()

      (* Remove CCLINK from TERM's parent's dll.
       * If TERM's parent list becomes empty, it's dead, too, so free it.
       *)
      and delPar(term, cclink) =
          case DL.remove cclink of (* Returns the dll elts before & after cclink. *)
              (DL.NIL, after) => let val parref = termParRef term
                                 in parref := after;
                                    case after of DL.NIL    => free term
                                                | DL.Node _ => ()
                                 end
            | _ => ()

  in free node
  end

(* Replace one child w/another in the tree.
 * - OLDPREF is the parent dll for some term -- the old term.
 * - NEW is the replacement term.
 * Add each element of the dll !OLDPREF to NEW's parent list. Each such
 * element indicates some parental downlink; install NEW in the right slot
 * of the indicated parent. When done, set OLDPREF := NIL.
 *
 * Actually, we don't move the dll elements over to NEW's parent list one at
 * a time -- that involves redundant writes. E.g., if !OLDPREF is 23 elements
 * long, don't move the elements over one at a time -- they are already nicely
 * linked up. Just connect the last elt of !OLDPREF & the first element of
 * NEW's existing parent list, saving 22*2=44 writes. Because it physically
 * hurts to waste cycles.
 *)
fun replaceChild(oldpref, new) =
    let val cclinks = !oldpref
        val newparref = termParRef new

        fun installChild(LambdaBody(Lambda{body,...})) = body := SOME new
          | installChild(AppFunc(App{func,...}))       = func := SOME new
          | installChild(AppArg(App{arg,...}))         = arg  := SOME new
```

```
        fun lp(prev, prevnext, DL.NIL) =
            (prevnext := !newparref ;
             case !newparref of DL.NIL           => ()
                              | DL.Node(p, _, _) => p := prev)
          | lp(prev, prevnext, node as DL.Node(_,cc, n as ref next)) =
              (installChild cc; lp(node, n, next))

    in case cclinks of DL.NIL => ()
                     | node as DL.Node(_,cc,n as ref next) =>
                       (oldpref := DL.NIL; installChild cc;
                        lp(node, n,next); newparref := cclinks)
    end


(* Allocate a fresh lambda L and a fresh var V. Install BODY as the body of
 * the lambda -- L points down to BODY, and L is added to BODY's parent list.
 * The fresh var's name (semantically irrelevant, but handy for humans) is
 * copied from oldvar's name.
 *
 * Once this is done, kick off an OLDVAR->V upcopy to fix up BODY should it
 * contain any OLDVAR refs.
 *)
fun newLambda(oldvar, body) =
    let val Var{name, parents = varparents, ...} = oldvar
        val var = Var{name = name,
                      uniq = newUniq(),
                      parents = ref DL.NIL}
        val bodyRefCell = ref NONE
        val ans = Lambda{var     = var,
                         body    = ref(SOME body),
                         bodyRef = bodyRefCell,
                         uniq    = newUniq(),
                         parents = ref DL.NIL}
        val cclink = DL.new(LambdaBody ans)
    in bodyRefCell := SOME cclink;
       addToParents(body, cclink);
       (* Propagate the new var up through the lambda's body. *)
       DL.app  (upcopy (VarT var))  (!varparents);
       LambdaT ans
    end

(* Allocate a fresh app node, with the two given params as its children.
 * DON'T install this node on the children's parent lists -- see "a subtle
 * point" above for the reason this would get us into trouble.
 *)
and newApp(func, arg) =
    let val funcRef = ref NONE
        val argRef  = ref NONE
        val app     = App{func    = ref(SOME func),
                          arg     = ref(SOME arg),
                          funcRef = funcRef,
                          argRef  = argRef,
                          copy    = ref NONE,
                          parents = ref DL.NIL,
                          uniq    = newUniq()}
    in funcRef := SOME( DL.new(AppFunc app) );
       argRef  := SOME( DL.new(AppArg  app) );
       app
    end

(* upcopy newChild parRef -> unit
 ********************************************************************************
 * The core up-copy function.
 * parRef represents a downlink dangling from some parent node.
 * - If the parent node is a previously-copied app node, mutate the
```

```
 *   copy to connect it to newChild via the indicated downlink, and quit
 * - If the parent is an app node that hasn't been copied yet, then
 *   make a copy of it, identical to parent except that the indicated downlink
 *   points to newChild. Stash the new copy away inside the parent. Then take
 *   the new copy and recursively upcopy it to all the parents of the parent.
 * - If the parent is a lambda node L (and, hence, the downlink is the
 *   "body-of-a-lambda" connection), make a new lambda with newChild as
 *   its body and a fresh var for its var. Then kick off an upcopy from
 *   L's var's parents upwards, replacing L's var with the fresh var.
 *   (These upcopies will guaranteed terminate on a previously-replicated
 *   app node somewhere below L.) Then continue upwards, upcopying the fresh
 *   lambda to all the parents of L.
 *)

and upcopy newChild (LambdaBody(Lambda{var, parents,...})) =
    DL.app (upcopy (newLambda(var, newChild))) (!parents)

  (* Cloning an app from the func side *)
  | upcopy new_child (AppFunc(App{copy as ref NONE, arg, parents, ...})) =
    let val new_app = newApp(new_child, valOf(!arg))
    in copy := SOME new_app;
       DL.app (upcopy (AppT new_app)) (!parents)
    end

  (* Copied up into an already-copied app node. Mutate the existing copy & quit. *)
  | upcopy newChild (AppFunc(App{copy = ref(SOME(App{func,...})), ...})) =
    func := SOME newChild

  (* Cloning an app from the arg side *)
  | upcopy new_child (AppArg(App{copy as ref NONE, func, parents, ...})) =
    let val new_app = newApp(valOf(!func), new_child)
    in copy := SOME new_app;
       DL.app (upcopy (AppT new_app)) (!parents)
    end

  (* Copied up into an already-copied app node. Mutate the existing copy & quit. *)
  | upcopy newChild (AppArg(App{copy = ref(SOME(App{arg,...})),...})) =
    arg := SOME newChild


(* Contract a redex; raise an exception if the term isn't a redex. *)

fun reduce(a as App{funcRef, func = ref(SOME(LambdaT l)),
                    argRef,  arg  = ref(SOME argterm),
                    parents, ...}) =
    let val Lambda {var, body, bodyRef, parents = lampars, ...} = l
        val Var{parents = vpars as ref varpars, ...} = var
        val ans = if len1(!lampars)

                  (* The lambda has only one parent -- the app node we're
                   * reducing, which is about to die. So we can mutate the
                   * lambda. Just alter all parents of the lambda's vars to
                   * point to ARGTERM instead of the var, and we're done!
                   *)
                  then (replaceChild(vpars, argterm);
                        valOf(!body))

                  (* Fast path: If lambda's var has no refs,
                   * the answer is just the lambda's body, as-is.
                   *)
                  else if varpars = DL.NIL then valOf(!body)

                  (* The standard case. We know two things:
                   * 1. The lambda has multiple pars, so it will survive the
                   *    reduction, and so its body be copied, not altered.
```

```
                 * 2. The var has refs, so we'll have to do some substitution.
                 *    First, start at BODY, and recursively search down
                 *    through as many lambdas as possible.
                 *
                 * - If we terminate on a var, the var is our lambda's var,
                 *   for sure. (OTW, #2 wouldn't be true.) So just return
                 *   BODY back up through all these down-search lambda-
                 *   skipping calls, copying the initial lambdas as we go.
                 * - If we terminate on an app, clone the app & stick the
                 *   clone in the app's copy slot. Now we can do our VAR->ARG
                 *   up-copy stuff knowing that all upcopying will guaranteed
                 *   terminate on a cached app node.
                 *
                 * When we return up through the initial-lambda-skipping
                 * recursion, we add on copies of the lambdas through
                 * which we are returning, *and* we also pass up that top
                 * app node we discovered. We will need it in the
                 * subsequent copy-clearing phase.
                 *)
                else let fun scandown(v as VarT _) = (argterm,NONE) (* No app! *)

                           | scandown(l as LambdaT(Lambda{body,var,...})) =
                             let val (body',topapp) = scandown(valOf(!body))
                                 val l' = newLambda(var, body')
                             in (l', topapp)
                             end

                           | scandown(AppT(a as App{arg,func,copy,...})) =
                             (* Found it -- the top app.          *)
                             (* Clone & cache it, then kick off a *)
                             (* var->arg upcopy.                  *)
                             let val a' = newApp(valOf(!func), valOf(!arg))
                             in copy := SOME a';
                                DL.app (upcopy argterm) varpars;
                                (AppT a', SOME a)
                             end

                         val (ans, maybeTopApp) = scandown (valOf(!body))

                         (* Clear out the copy slots of the app nodes. *)
                         in case maybeTopApp of
                                NONE => ()
                              | SOME app => clearCopies(l,app);
                            ans
                         end


    (* We've constructed the contractum & reset all the copy slots. *)

    in replaceChild(parents, ans);    (* Replace redex w/the contractrum. *)
       freeDeadNode (AppT a);         (* Dealloc the redex.               *)
       ans                            (* Done.                            *)
    end


(* Call-by-name reduction to weak head-normal form. *)
fun normaliseWeakHead(AppT(app as App{func, arg, ...})) =
      (normaliseWeakHead(valOf(!func));
       case valOf(!func) of LambdaT _ => normaliseWeakHead(reduce app)
                          | _         => ())
  | normaliseWeakHead _ = ()

(* Normal-order reduction to normal form. *)
fun normalise(AppT(app as App{func, arg, uniq,...})) =
      (normaliseWeakHead(valOf(!func));
```

```
    case valOf(!func) of LambdaT _ => normalise(reduce app)
                       | VarT _    => normalise(valOf(!arg))
                       | app'      => (normalise app';
                                       normalise(valOf(!arg))))
| normalise(LambdaT(Lambda{body,...})) = normalise(valOf(!body))
| normalise _ = ()
```