Bottom-up β-reduction: uplinks and λ-DAGs

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

Mitchell Wand
Northeastern University
wand@ccs.neu.edu

Abstract

If we represent a λ-calculus term as a DAG rather than a tree, we can efficiently represent the sharing that arises from β-reduction, thus avoiding combinatorial explosion in space. By adding uplinks from a child to its parents, we can efficiently implement β-reduction in a bottom-up manner, thus avoiding combinatorial explosion in time required to search the term in a top-down fashion.

We present an algorithm for performing β-reduction on λ-terms represented as uplinked DAGs; describe its proof of correctness; discuss its relation to alternate techniques such as Lamping graphs, the suspension λ-calculus (SLC) and director strings; and present some timings of an implementation.

Besides being both fast and parsimonious of space, the algorithm is particularly suited to applications such as compilers, theorem provers, and type-manipulation systems that may need to examine terms in-between reductions—*i.e.*, the "readback" problem for our representation is trivial.

Like Lamping graphs, and unlike director strings or the suspension λ-calculus, the algorithm functions by side-effecting the term containing the redex; the representation is *not* a "persistent" one.

The algorithm additionally has the charm of being quite simple; an appendix contains a complete, annotated implementation of the core data structures and algorithms in 180 lines of SML.

1 Introduction

The λ-calculus [2, 5] 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. Here, our particular interest is in the needs of client applications such as compilers, which may use λ-terms to represent both program terms as well as complex types. We are somewhat less focussed on the needs of graph-reduction engines, where there is greater representational license—a graph reducer can represent a particular λ-term as a chunk of machine code (*e.g.*, by means of supercombinator extraction), because its sole focus is on *executing* the term. A compiler, in contrast, needs to examine, analyse and transform the term in-between reductions, which requires the actual syntactic form of the term be available at the intermediate steps.

Of the three basic operations on terms in the λ-calculus—α-conversion, β-reduction, and η-reduction—it is β-reduction that accomplishes the "heavy lifting" of term manipulation. (The other two operations are simple to implement.) Unfortunately, naïve implementations of β-reduction can lead to exponential time and space blowup.

There are only three forms in the basic language: λ expressions, variable references, and applications of a function to an argument:

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

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.)

β-reduction is the operation of taking an application term whose function subterm is a λ-expression, and substituting the argument term for occurrences of the λ'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

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

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 λ-expression $\lambda y.b$, we must first replace the λ-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 λ-expression binds a unique variable.

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

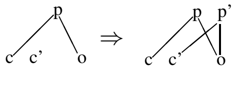
2 Guided tree substitution

Let's turn to a simpler task to develop some intuition. Consider inserting an integer into a 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 β-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:



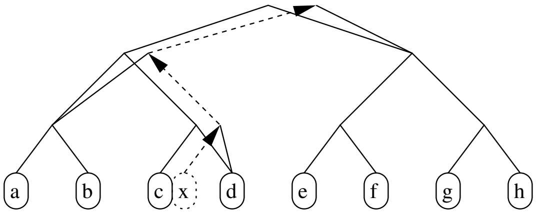


Figure 2: Replacing a single leaf in a binary tree by following uplinks. Here, we make a copy of the original tree, replacing leaf c with x. Dotted lines show the path of algorithm as it copies up the spine of the tree from c to the root.

```
while old $\neq$ 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 β-reduction on λ-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 λ-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 λ-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 {
          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 $\langle$*parent*, *relation*$\rangle$ pair. For example, if node c is the left child of node p, then the pair $\langle p$, 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 β-reduction algorithm in place, and can fill in the details specific to our λ-expression domain.

Basic representation We will represent a λ-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 λ-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 β-reduction operation. Note that this amounts to an α-uniqueness condition—we require that every λ-expression bind a unique variable.

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

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

Copying λ-expressions With all the above structure in place, the algorithm takes shape. To perform a β-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. λ-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 λ parent of c: $\lambda y.c$. The general structure of the algorithm calls for us to make a copy of the λ-expression, with body c'. But we must also allocate a fresh variable, y', for our new λ-expression, since we require all λ-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 β-reduction, the original redex must die. That is, the model we have is that we start with a λ-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 "pure functional" or "persistent" 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 β-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 λ-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 λ-expression being reduced and scan downwards from λ-expression to body, until we encounter a non-λ-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 λ-expressions, wrapping fresh λ-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 λ-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 λ-expressions leading from a back to the top $\lambda x.b$ term. As we pass back up through each λ-expression $\lambda y.t$, we allocate a fresh λ-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 λ-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 λ-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 λ-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 λ-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 λ-expression binding that variable. So if the λ-expression has no parents, then all paths upwards from its variable must terminate at the λ-expression itself.

This opens up the possibility of an alternate, fast way to produce the contractum: when the λ-expression being reduced has only one parent, mutate the λ-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 λ-expression's body as our answer, discarding the λ-expression and its variable x (in general, a λ-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 λx node has other parents besides application 1—so we cannot blindly mutate it at will, without corrupting what the other parents see. (If the λx node *doesn't* have other parents, then the single-reference fast-path described in the previous section applies, and we *are* allowed to mutate the the term, for a very fast reduction.)

In the following subfigure, 5(b), we focus in on the body of the λ-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 λy 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, λy, 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 λy, 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 λy 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 λy term that kicked off the $y \mapsto y'$ copy back in subfigure 5(f).

In subfigure 5(h), the λy 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 λy 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. A precise, fully-detailed argument establishing its correctness 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. We will take λ-terms to be graphs: a term is a finite DAG with labelled edges and nodes. Every vertex is labelled as either a λ, an application, or a variable-reference vertex. We will frequently use l for λ 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) body edge beginning at node l. A λ node is connected to its body by a body edge; and to its bound variable by a bvar edge. An application node is connected to its function by a fun edge; and to its argument by an arg edge. We assume enough constraints on labelling and graph structure to preserve the syntactic structure of the λ-calculus, *e.g.*,

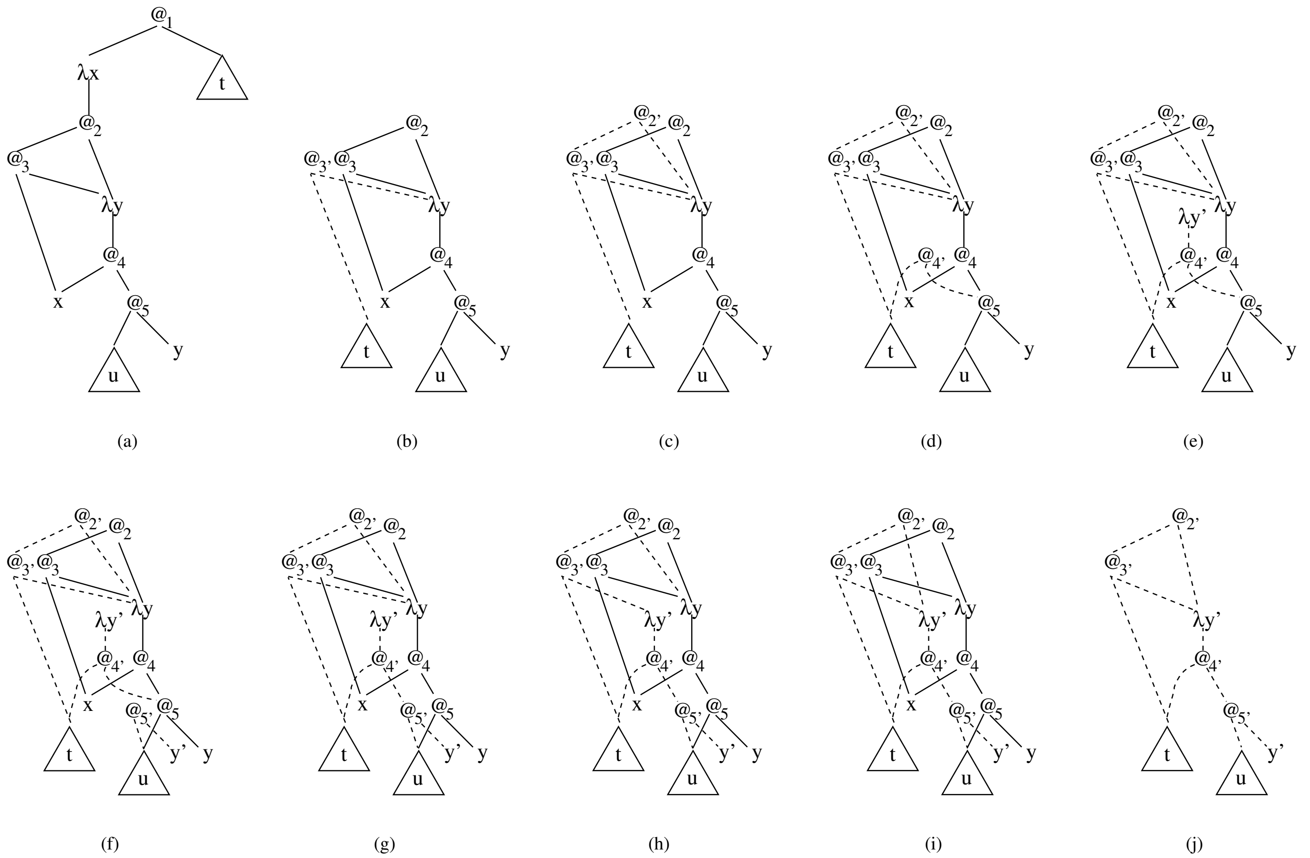


Figure 5: A trace of a bottom-up reduction of the term $(\lambda x.(x(\lambda y.x(uy)))(\lambda y.x(uy)))t$, where the $(\lambda y.x(uy))$ term is shared, and sub-terms t and u are not specified.

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 λ vertex binding that variable.

Define Λ_{dag} to be the set of DAGs satisfying these constraints. The classic λ-calculus is defined in terms of trees; define Λ_{tree} to be the standard realisation of λ-terms as abstract-grammar trees, with variables drawn from the variable-labelled vertices of Λ_{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, unfold : $\Lambda_{\text{dag}} \to \Lambda_{\text{tree}}$.

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

(Here, we take "$n_1 \;@\; n_2$" 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.

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))
```

Punfold takes a graph, a root vertex n, and a substitution σ. We model a substitution as a partial function from (variable) nodes to Λ_{tree} terms:

$$\sigma \in \text{Subst} = \text{Node} \rightharpoonup \Lambda_{\text{tree}}.$$

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). Punfold recursively unfolds new structure, but when it traces into old structure, it unfolds *and then applies* σ to the old sub-DAG—that is, Punfold "predicts" what the algorithm will do when it gets

```
Sub(g,$old$,$new$,$top$,$\sigma$,$ip$) =
  iter $\lambda$(par,rel) g . uc(new,par,rel, top,$\sigma$,$ip'$,g)
       g
       parlinks(old,$g_0$)
  where $ip'$ = $ip \cup \{old\}$
        iter f g $\emptyset$ = g
        iter f g $(\{x\} \cup S)$ = iter f (f x g) S


uc(new,l,body,top,$\sigma$,$ip$,g) =
  (* Upcopy into $\lambda$ node l along body edge *)
  b = l.body        be = $l \xrightarrow{\text{body}} b$
  y = l.bvar        y' = freshnode var
  g += $(\{y'\},\ \emptyset,\ \{(y,y')\},\ \{be\})$
  g = Sub(g, y, y', b, $[y \mapsto y']\sigma$, $ip$)
  l' = freshnode $\lambda$
  g += $(\{l'\},\ \{l' \xrightarrow{\text{bvar}} y',\ l' \xrightarrow{\text{body}} new\},\ \{(l,l')\},\ \emptyset)$
  return Sub(g, l, l', top, $\sigma$, $ip$)
```

```
uc(new, a, fun, top, $\sigma$, $ip$, g) =
  (* Upcopy into app node a along fun edge *)
  g.T += $\{a \xrightarrow{\text{fun}} a.\text{fun}\}$
  if (a,a') $\in$ $g.\rho$ then
      replace $a' \xrightarrow{\text{fun}} n$ edge
        with $a' \xrightarrow{\text{fun}} new$ edge
        in g
      return g
  else
      a' = freshnode app
      g += $(\{a'\},\ \{a' \xrightarrow{\text{fun}} new, a' \xrightarrow{\text{arg}} a.\text{arg}\},\ \{(a,a')\},\ \emptyset)$
      return Sub(g, a, a', $top$, $\sigma$, $ip$)

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

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

to that sub-DAG. 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. (Note that Punfold passes σ down past binding occurrences as it recurses. This is safe to do, as we will restrict the domain of σ to be old structure, so that passing these substitutions into the scope of forms that bind new variables cannot cause capture.)

With this structure in place, the game plan should now begin to emerge. Punfold$(g_0, b, [x \mapsto n])$ produces the contractum for D