An Algorithm for Optimal Lambda Calculus Reduction

John Lamping

Published in Seventeenth ACM Symposium on Principles of Programming Languages, pages 16-30, 1990.

An Algorithm for Optimal Lambda Calculus Reduction

John Lamping

January 17, 1990

We present an algorithm for lambda expression reduction that avoids any copying that could later cause duplication of work. It is optimal in the sense defined by Lévy. The basis of the algorithm is a graphical representation of the kinds of commonality that can arise from substitutions; the idea can be adapted to represent other kinds of expressions besides lambda expressions. The algorithm is also well suited to parallel implementations, consisting of a fixed set of local graph rewrite rules.

Overview

The lambda calculus[1] defines beta reduction (reducing the application of a lambda term to an argument) in terms of substitution; each occurrence of the variable bound by the lambda is replaced by a copy of the argument. This can create extra work, since any work needed to simplify the argument must be repeated on each copy.

Lévy[4, 5] has shown that there are lambda expressions for which any order of reduction duplicates work. One example is

```
((λg.(g (g (λx.x))))
 (λh.((λf.(f (f (λz.z))))
      (λw.(h (w (λy.y)))))))
```

which has two redexes, an outer one ((λg ...) ...), and an inner one ((λf ...) ...). If the outer redex is reduced first the inner redex will be duplicated, and each copy will have to be reduced. On the other hand, if the inner redex is reduced first, its argument, (λw.(h (w (λy.y)))), will be duplicated, which will cause redundant work later once a value for h is determined. Four copies of the application (h (w (λy.y))) will ultimately need to be simplified if the inner redex is reduced first, compared to only two copies if the outer redex is reduced first, one for each different value of h.

In general, a tension can occur when a function that is used several times, (λh ...) in the example above, contains a subfunction that it uses several times, (λw.(h (w (λy.y)))) in the example above. The tension occurs, as in this case, when the subfunction can be simplified given either the value of the argument to the function or the value of the argument to the subfunction. It is possible to simplify each use of the subfunction (to fold it inline) before any of the uses of the function are simplified, so that the work of simplifying each use of the subfunction can be shared among all uses of the function. Alternatively, it is possible to simplify the subfunction (to partial evaluate it) each time the main function is used so that that work of simplifying the subfunction can be shared among all uses of the subfunction. But whichever simplification is done first ends up duplicating the work of the other.

The technique of graph reductions[9] avoids some copying by treating a lambda expression as a tree, which is represented by an acyclic directed graph. Lambda calculus reductions are then modeled by graph operations. Since the lambda expression is represented by a graph, identical subtrees (identical subexpressions) can be represented with a single piece of graph. In particular, reducing a redex doesn't require copying the argument; each use of the argument simply gets a link to the original. This can save work later because one simplification step in a shared section of the graph corresponds to what would be multiple simplifications in the represented lambda expression.

But this only postpones copying by one step. If the lambda term in a redex is represented by a shared section of graph, that section must be copied before the redex can be reduced. Otherwise, the substitution called for by the reduction would affect not only the body of the lambda term in the redex, but also the bodies of the other lambda terms represented by the shared piece of graph. In the example above, graph reduction can reduce the original two redexes without doing any copying, but then it will have to copy in order to reduce any of the newly formed redexes, the applications of the shared functions (λh ...) and (λw.(h (w (λy.y)))). It will duplicate work in doing so.

It is possible to combine graph reduction with the translation of lambda expressions into combinators[8]. Combinator reductions are used to simpify the translated graph instead of the beta conversion rule; a single beta reduction becomes modeled by a series of combinator reductions, each of which is a simple local graph operation. Where a beta reduction on a graph would require copying a function body, the corresponding combinator reductions will be able to avoid copying any subexpressions that

don't contain the variable bound by the lambda. This, however, isn't enough to eliminate all duplication of work. In the example, combinator reduction would reduce the outer redex first and then would have to partially copy combinators corresponding to the (λh ...) expression. It would not have to copy the (λf ...) subexpression, because it contains no h, but combinator reduction would have to copy the application ((λf ...) ...), because the argument contains a free h. In copying that application, it duplicates work.

Lévy has given a precise definition of what it would mean to never duplicate work. Given a tree that represents a lambda expression, he defines a system for labelling its links, and defines how the labels should be transformed by lambda reduction. Two links will get the same labelling under these rules if they have a common origin (where having the same origin includes having resulted from copies of a single link or having resulted from reductions of copies of a single redex). Using this labelling, Lévy defines a parallel reduction operation which reduces in one step all redexes having a given label on the link between their application and lambda, thus reducing all copies of a redex in one step. He shows that if this parallel reduction is considered to be a unit step, then no order of reduction duplicates work. From this it follows that normal order parallel reduction is optimal. Levy did not, however, show how to implement parallel reductions.

Staples[6, 7] has developed an algorithm designed to implement Lévy's parallel reductions, based on using graphical structures to represent partial substitution environments. But his algorithm appears to accumulate structure monotonically as it executes and there doesn't appear to be a bound on the amount of work the algorithm must do for each reduction. Kathail[3] has recently developed an implementation of parallel reductions, but hasn't yet finished a description of it, and so we have not yet been able to compare it to the algorithm presented here.

We have developed an algorithm based on an extension of graph reduction. Ordinary graph reduction makes use of fan-in: several links coming into a single subgraph, each corresponding to a different instance of the subgraph; our algorithm also makes use of fan-out: several links coming out of a single subgraph, each corresponding to a different instance of the subgraph. This algorithm is able to avoid copying in the situations outlined above. More generally, the algorithm will always use a single piece of structure to represent redexes that are equivalent under Lévy's labelling. Although the presentation of the algorithm will be in terms of lambda calculus reductions, the essence of the algorithm is the graphical structure that represents the kinds of commonality that can result from substitution. This is adaptable to other situations involving substitution.

The algorithm consists of a fixed set of graph rewrite rules, which specify local incremental modifications to the graph. The decision of which rule to apply to the graph next and where to apply it is made non-deterministically. No order of rule execution will give incorrect results or generate duplicate beta reductions, but the most efficient implementations would give some rules priority over others. Since the rules operate locally, several rule executions can be done in parallel, on different parts of the graph.

While the algorithm avoids copying, it does not merge identical substructures. That is, if it happens during the course of lambda reduction that two initially different parts of a lambda expression are reduced to identical expressions, the algorithm won't notice that. It will reduce both expressions independently. A facility to notice identical subexpressions could be added to the algorithm, but it wouldn't supplant the algorithm, because the algorithm is able to preserve sharing between subexpressions, like (g (h x)) and (g (h y)), that have no common subexpressions but still have some structure in common.

The rest of this paper illustrates the algorithm in action, gives its rewrite rules, and sketches proofs of its correctness and optimality. The complete rewrite rules of the algorithm appear in figures 3, 4, and 5 at the end of the paper, but are very difficult to understand in isolation. Rather than focus on individual rules, we illustrate the effects of the rules on the graph structure, first with a simplified version of the algorithm, and then with the full algorithm. The simplified version leaves one crucial issue unresolved, which the full algorithm addresses. The paper concludes with the key definitions and lemmas in the proofs of correctness and optimality.

One discouraging note: while both an informal argument and experience with an implementation of the algorithm indicate that the amount of bookkeeping work the algorithm requires for each beta reduction step is proportional to the cost of doing a substitution, we haven't proven this. The difficulty appears to be in correctly formulating the bound.

Simplified Execution Example

Figure 1 illustrates the step by step execution of a simplified version of the algorithm. This section explains what is happening in the figure.

Like graph reduction, the algorithm treats a lambda expression as a tree, which it represents with a rooted graph. Graph A in figure 1 shows the graph the algorithm uses to represent the lambda expression

```
((λg.(g (g (λx.x))))
 (λh.((λf.(f (f (λz.z))))
      (h (λy.y)))))
```

The nodes in the graph fall into two categories. The ordinary nodes represent parts of a lambda expression. These are the application nodes, written @; lambda nodes, written λx; and variable nodes, written x. The control nodes, on the other hand, don't represent parts of a lambda expression, but instead control how the graph represents a tree. Here, these are the fan nodes, written $\overset{\star}{\bigtriangledown}^{\circ}$. This graph resembles the one that a standard graph reduction algorithm would use to represent the same lambda expression. The obvious difference is the two fan nodes, which explicitly show where sharing is occurring.

Throughout figure 1, the thickly drawn link in each graph indicates where the next rule execution occurs. In graph A, this is the topmost application, where the rule which simulates beta reductions (rule I.a in figure 3) will be executed, resulting in graph B. The rule indicates that a subgraph that matches the left hand side of the arrow should be replaced by an instance of the right hand side of the arrow. Node variables, notated like ⓐ, match any node. On the right hand side, they show how the resulting subgraph should be connected with the rest of the graph. This particular rule eliminates nodes and changes connections; other rules will create nodes as well. For a formal semantics of a similar graph rewrite rule system, see Barendregt et al[2].

The rule presumes that all the variable occurrences bound by a given lambda are represented by a single variable node, one variable node per lambda node. This property means that the beta reduction rule doesn't have to do any copying; it can just connect the argument to where the variable was, as illustrated in the figure. The property is first established when a lambda expression is translated into a graph by establishing one variable node for each lambda node, using fan nodes to consolidate the different variable instances bound by a lambda into one variable node.

There is a technical problem in making sure that the variable node matched by the rule is the mate of the lambda node matched by the rule; we resolve this by assuming that there is a link in the graph between the lambda node and its corresponding variable node, indicated in the picture by their common variable name rather than by a line. This also means that the left hand side of the rule is, in fact, connected.

Another execution of rule I.a, this time on the application of the $(\lambda f \ldots)$ yields graph C, which represents the lambda expression

$$
\begin{array}{l}
((\lambda h.((h\ (\lambda y.y))\\
\qquad ((h\ (\lambda y.y))\\
\qquad\ (\lambda z.z))))\\
\ ((\lambda h.((h\ (\lambda y.y))\\
\qquad ((h\ (\lambda y.y))\\
\qquad\ (\lambda z.z))))\\
\ (\lambda x.x)))
\end{array}
$$

where the subexpression $(h\ (\lambda y.y))$ occurs four times, although it occurs only once in the graph.

So far, the algorithm has done nothing different from what graph reduction would do; the next rule execution will change that. The only redexes in the lambda expression are applications of one of the copies of $(\lambda h \ldots)$. Since those copies are represented by a shared piece of graph, ordinary graph reduction would have to copy the shared piece before it was possible to proceed with a beta reduction.

The algorithm takes the more subtle step that results in graph D. Rule II.c in figure 3 replaces a shared lambda by two lambdas with a shared body; it replaces the variable node paired with the original lambda node with a fan node leading to two variable nodes for the two lambda nodes (ignore the annotation i on the nodes in the rule for the moment). The "upside down" fan node is no different in kind from the other fan nodes; it is just more convenient to draw it in that orientation. The node type is independent of orientation because the graph isn't directed; although application and lambda nodes will always be oriented the same way with respect to the root of the graph, the control nodes can occur in either orientation. The definition of the graph and the operation of the rules only consider which sites of each node are connected to which sites of the other nodes, with no notion of direction. (It would be possible to define an equivalent algorithm on directed graphs, but the distinctions imposed by the directedness of the links would necessitate gratuitously repetitious node types and rules.)

Starting from the perspective of the root of the graph, however, it is natural to superimpose a sense of direction on the graph and to distinguish an "upside down" fan node from a "right side up" fan node. From this theoretical perspective, an "upside down" fan functions as a fan-out, rather than as a fan-in. To see the meaning of a fan-out, first notice that in a graph without fan-outs, every path from the root that follows the superimposed sense of direction to reach an ordinary node represents a path in the lambda expression tree (equivalently, a node in the lambda expression tree, since there is a one-to-one pairing in trees between nodes and paths from the root). The same is true in a graph with fan-outs, except that there is a restriction on the paths. Every fan-out along a path will be paired with some fan-in along the path, and the path must follow the branch out of the fan-out that is marked the same ($\star$ or $\circ$) as the branch it followed into the fan-in. Thus, graph D stands for the lambda

expression

```
((λh.((h (λy.y))
      ((h (λy.y))
       (λz.z))))
 ((λh'.((h' (λy.y))
        ((h' (λy.y))
         (λz.z))))
  (λx.x)))
```

because the fan-out is paired with the top fan-in, which means that paths from the λh, which go in the ⋆ branch of the top fan-in, must go out the ⋆ branch of the fan-out to the h variable; and correspondingly for paths from the λh'.

If, on the other hand, the fan-out were paired with the lower fan-in, the graph would stand for the lambda expression

```
((λh.((h (λy.y))
      ((h' (λy.y))
       (λz.z))))
 ((λh'.((h (λy.y))
        ((h' (λy.y))
         (λz.z))))
  (λx.x)))
```

which has an h and h' incorrectly interchanged. Determining which fan-outs pair with which fan-ins is a crucial issue, but that is the issue that is not addressed by this simplified version of the algorithm. The full algorithm uses several additional kinds of control nodes to delimit scopes within which fan-ins and fan-outs can pair. For this simplified explanation, we have omitted those control nodes and the rules that manipulate them. They account for about half of the rule executions of the full algorithm. For the moment, we will assume that the pairing of fan-ins and fan-outs is determined omnisciently.

Finally, it is important to note that the notion of paths is part of an explanation of what lambda expression a graph stands for, not something which the algorithm is directly sensitive to. The algorithm simply does local operations, which are in accord with the bigger picture without being directly cognizant of it.

After the λh node has been split, the beta reduction rule is applicable again. Applying it to reduce the application of (λh' ...) results in graph E.

Various rules are applicable at this point; assume that the algorithm turns to the bottom application node in graph E, which represents applications of two different functions: h, and (λx.x). To make progress, it is necessary to duplicate the application node, so the two functions can be reduced separately. This is the only situation where the algorithm duplicates an application node: when the node represents applications of different functions. This stipulation means that an application node is not duplicated unless the resulting nodes will represent different work; it is the key to the optimality of the algorithm. Graph F shows the result of rule IV.e, which duplicates the application; the two new applications get their respective functions and share their common argument. The new graph represents the same lambda expression.

Again, various of the algorithm's rules apply at various points of this graph. Assuming the algorithm chooses to reduce the application of (λx.x), the result is shown in graph G. Next, the rule for duplicating a shared lambda node can be applied to the λy node, giving graph H. In this case, the common body shared between the two new lambda nodes is a single edge. Graph I shows what then happens when the lower fan-out meets its paired fan-in: rule V.b shows how they annihilate each other and connect corresponding links, ⋆ with ⋆ and ∘ with ∘. This operation leaves the graph representing the same lambda expression as before; the lambda expression tree represented by the middle graph only reflected paths that went through corresponding links of the fans, and those two valid routes are replaced by two links in the right hand graph.

There is still a fan-out meeting a fan-in, but these fans aren't paired with each other, the fan-out is instead paired with the upper fan-in. All four routes going through the two fans contribute to the lambda expression

```
((λh.((h (λy.y))
      ((h (λy.y))
       (λz.z))))
 ((λy'.y')
  ((λy'.y')
   (λz.z))))
```

represented by the graph, so they must all be preserved. Rule V.a shows how the fans should duplicate each other, resulting in graph J, which has a new link for each of the four routes through the two fans in the former graph. The mechanism for deciding whether to use rule V.a or rule V.b will be taken up in the next section.

The algorithm proceeds through the rest of figure 1 using the five rules already illustrated. A few more steps beyond the figure finally result with (λz.z).

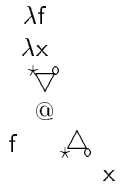
Execution Example

One of the key points of the algorithm is that a subpiece of the graph which contains fan-outs can represent different lambda expressions for different paths which reach it. For example, the graph segment

which arose during the simplified example, represents either h or h', depending on the history of the path that reaches it. Any path from the root can be thought of as accumulating a context that records which branches of fan-ins it traversed, so that it knows which branches of fan-outs to take. The context combines with the structure of a piece of graph to determine which lambda expression is represented. Like the notion of paths, this notion of context is not a part of the algorithm, but rather a tool to analyze and understand the algorithm.

The context must not only record which branches of fan-ins were taken, but must organize that information so that it is possible to determine which fan-ins pair with which fan-outs. This section deals with the issue of how to do that.

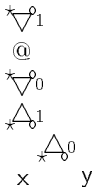
An obvious idea is to label the fans in the graph, with fans paired if and only if they have identical labels. To see why this scheme isn't adequate, first notice that graphs containing looping paths, for example



can arise when one instance of a shared expression gets substituted inside another instance of that expression. This graph represents the lambda expression (λf.(λx.(f (f x)))) and is the result, for example, after the algorithm beta reduces

(λf.(λx.((λg.(g (g x)))
(λy.(f y)))))

Looping paths present a problem in situations where a looping path traverses the same fan-in several times before traversing its paired fan-out, such as in the graph segment:



It then becomes necessary to determine which traversal of a fan-in should be paired with a particular traversal of a fan-out. Assume in the graph that the fan-in labeled 0 is paired with the fan-out labeled 0, and similarly for the fans labeled 1. Any legal path that reaches the fan-out labeled 0 will have gone through the fan-in labeled 0 twice. If the traversal of the fan-out should be paired with the second traversal of the fan-in then the graph segment represents the expression ((x y) (x y)). The segment can occur with this interpretation, for example, near the end of the simplification of

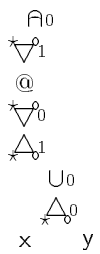
(λx.(λy. ((λf.((λh.(h (λp.(h (λq.q)))))
(λl.(((f (λn. (l n))) x) y))))
(λg.(λu.(λv.((g u) (g v))))))))

On the other hand, if the traversal of the fan-out should be paired with the first traversal of the fan-in then the graph segment represents the different expression ((x x) (y y)). The segment can occur with this interpretation, for example, near the end of the simplification of

(λx.(λy. ((λf.((λh.(h (λp.(h (λq.p)))))
(λl.(((f (λn. (l n))) x) y))))
(λg.(λu.(λv.((g u) (g v))))))))

The labels on the fans aren't able to distinguish the two cases.

Our solution adds a notion of enclosure to delimit the interaction of fans along paths. Specifically, traversals of fans from different enclosures along a path never pair. Without any intervening enclosures, adjacent identically labeled fans pair, so that the above graph represents the expression ((x y) (x y)). But bracket nodes (to be explained shortly) can be added to the graph:



so that the second traversal of the fan-in labeled 0 is enclosed, which makes the fan-out labeled 0 pair the first traversal of the fan-in; the graph represents the expression ((x x) (y y)).

For an explanation of brackets, we return to the example of the previous section. The full algorithm represents the initial lambda expression

((λg.(g (g (λx.x))))
(λh. ((λf. f (f (λz.z)))
(h (λy.y)))))

with graph A of figure 2, which is identical to the graph of the simplified example, except that each fan is in its own enclosure, represented in the graph by two new kinds of control nodes, ∪, and ⊎, called bracket nodes, which indicate enclosure boundaries. (For the moment, ignore

the difference between the two kinds of bracket nodes and ignore the number next to all control nodes in the graph). The bracketing nodes can be viewed as delimiting enclosures, which indicate how fans pair; or they can be viewed as serving to organize the context and to control its accumulation, which will then determine how fans pair. The two circles on the illustration of graph A are not an actual part of the graph structure, but serve as an aid to visualizing the enclosures (It isn't always possible to illustrate such boundaries on a graph, because brackets enclose segments of paths, not segments of graphs. For the sample execution, the two will coincide, but that wasn't the case in the graph above). The open end of a bracket node points toward the inside of the enclosure it is a part of. From the point of view of a path starting at the root of the graph, a $\cap$ node looks like an open bracket; crossing it puts the path inside a new enclosure. Similarly, a $\cup$ looks a close bracket. As will become apparent later in the example, enclosures can nest or overlap, and a single enclosure can be composed of disjoint regions.

The effect of the brackets on the context accumulated by a path can be deduced by noting that since the fans inside an enclosure cannot pair with fans outside the enclosure, once a path leaves an enclosure its context doesn't reflect any enclosed fans; from outside, it can never matter which branches were taken inside. This requires that the context look like a stack of separate frames or levels, one for each level of nesting of enclosures. Each level of the context records the branches taken through a sequence of fans all at the same level of nesting. An open bracket starts a new level of the context to reflect entering a new enclosure, and a close bracket discards the top level to reflect leaving that enclosure.

The brackets in the initial graph set up a crucial transparency property maintained by the algorithm: if a path segment in the graph corresponds in the lambda expression to going between a lambda and an occurrence of the variable it binds, then the control nodes along the path segment will have no net effect on the accumulated context. The initial bracketing accomplishes this by enclosing each fan, so that the effect of the fan on the context will be discarded. This is fine since, initially, no fan pairs with any other.

The transparency property is needed for the correctness of rule I.a, which simulates beta reduction. Recall that the rule disconnects the argument from the application and reconnects it where the bound variable node was. If the argument were to see different contexts in its new location than it did in its old, it would represent different lambda expressions after the beta reduction than it did before, and the transformation would have been incorrect. But consider any path to the application. The one step extensions of the path to the argument and to the lambda have the same context, since there are no intervening control nodes, and transparency ensures that any extensions of the path corresponding to variable occurrences bound by the lambda have that same context. So after the beta reduction rule runs, any path to the former argument will have the same context as the path in the original expression that represented the argument.

Two applications of rule I.a yield graph B in figure 2 (in this example, we will often show several rule executions at once). At this point, the simplified algorithm would have duplicated the λh node. But now there is a $\cup$ between the fan and the lambda node. There are a couple of ways of looking at what has to happen, each of which points out a problem.

The immediate goal is to get the $\cup$ node out of the way, presumably by moving it below the λh node, thus putting the lambda node and fan-in in the same enclosure. But in order to preserve the transparency property, it would also be necessary to add a node above the h node that undoes the effect of the $\cup$. The problem is that there is no way to undo the irreversible effect of a $\cup$ node. From the point of view of the context there is no way to know what information the node threw away; from the point of view of enclosures, the enclosure has been closed off and there is no way back in.

One step further ahead of this goal is the need to prepare for the duplication of the λh node. This will require the fan-in to be moved below the λh node and be paired with a fan-out placed where the h variable node is now. For the fans to be paired, they will have to be in the same enclosure. But that enclosure shouldn't include most of the body of the λh, otherwise the fans might incorrectly pair with some other fan inside the body (this isn't a problem in the current graph, but could be in general).

The solution is to have a disconnected enclosure that includes the fan-in and the λh node in one part, and the h node in the other. The first step is rule III.a, which rewrites the $\cup$ node to a new kind of control node, a conditional bracket, written $\sqcup$, as shown in graph C. A conditional bracket indicates that the area of the graph it encloses might be one component of a disconnected enclosure which has another component in the direction of the link through the bracket. This is indicated with a dashed line in illustrations of enclosures. As in this graph, some of the brackets around a region can be conditional brackets while others are unconditional brackets. Other parts of the enclosure to which the region belongs will never be found in the direction of links that cross unconditional brackets.

A conditional enclosure boundary doesn't necessarily mean that the area of graph it encloses has another component in the direction of the link through the bracket; that is why it is called conditional. For two areas to be part of the same enclosure, they must be at the same level of enclosure nesting and there must be a path segment

between them that enters both at conditional brackets. Put negatively, a conditionally enclosed area will never be part of the same enclosure as an unconditionally enclosed area or with an area at a different level of nesting. The rules reflect these principles by having back-to-back conditional brackets cancel each other (rule VII.c), while having a conditional bracket directly enclosed by an unconditional bracket turn into an unconditional bracket (rule VII.d).

In terms of the context, a conditional close bracket suspends a level of context. It says that the level it brackets should be encapsulated, but not thrown away. It acts rather like a closure-forming operator, wrapping the contents of the level it brackets into a capsule which is placed in the newly-uncovered next lower level. Inversely, a conditional open bracket expects to find a capsule as the most recent item of the level, which it opens up to form a new level. On the other hand if there are capsules on a level that a general close bracket discards, the capsules are discarded with everything else.

Changing a general bracket to a conditional bracket changes the contexts accumulated by paths and could thus potentially change the lambda expression represented by a graph, but the algorithm maintains an independence property to preclude that possibility in this case: the top level of the context will never influence which lambda expression is represented by the section of graph below a lambda node. This property means, essentially, that each lambda node has a level of context available for keeping track of different instantiations of the lambdas it represents, which won't interact with the graph below the lambda node. The property is sufficient to ensure the correctness of changing a close bracket above a lambda to a conditional close bracket, since the only difference in context that results from the change is an additional capsule in the top level, which can't affect the lambda expression represented by the graph.

Returning to the example, since the effect of a conditional bracket can be undone, rule II.a can move the conditional bracket below the λh node, putting its reverse above the h node to preserve the transparency of path segments from the λh node to the h node. The result, graph D, has an enclosure that consists of disconnected regions.

Now the lambda node is ready to be duplicated by rule II.c; the fan-out and fan-in would be in the same enclosure, and thus correctly pair with each other. But to stay on the topic of conditional brackets, but we will assume that the algorithm first deals with them some more. The algorithm tries to merge disconnected regions of a single enclosure by moving conditional brackets so that they enclose as large a region as possible; when the disconnected regions meet, the brackets can cancel, merging the regions. From graph D, the two conditional brackets can be propagated over applications by rules IV.b and IV.d, resulting in graph E. In each case a conditional bracket on one branch of the application turns into conditional brackets on each of the other two branches, so that the application becomes included in the enclosed region.

Now two of the conditional brackets have reached unconditional brackets. The conditional brackets are destined to meet and cancel, but there is another enclosure between them. In order for the conditional brackets to meet, at least one of them will have to go through that enclosure. The first step must be to transpose a conditional bracket past an unconditional bracket so that their respective enclosures overlap. But transposing the brackets without adjusting something else would change which close brackets close which open brackets, giving entirely rearranged enclosures, not the desired overlapping enclosures. The solution, implemented by rules VII.f and VII.h (with one more execution of IV.d thrown in) is shown in graph F, where the numbers that are attached to every control node are used to indicate overlapping enclosures.

A control node with a number i interacts with nodes i levels of enclosure removed. Thus the conditional bracket nodes in graph F with number 1 don't form an enclosure with the unconditional brackets they face, but rather with brackets one level of enclosure further out, forming the illustrated regions. Again, the outer two regions are two disconnected components of a single enclosure.

In terms of context, a control node with non-zero number doesn't act on the top level of the context, but rather acts as many levels down in the context as the value of its number (the top level being the 0th). The $\sqcup_1$ and $\sqcap_1$ nodes in graph F are acting not at the level of bracketing that directly encloses them, but rather at one level further out. In general, by keeping track of relative offsets of levels, the numbers make it possible to move enclosure boundaries past each other without getting them tangled.

Next, rule II.a moves one of the conditional brackets below the λy node while rule VI.f moves another above a fan-in, giving graph G. Enclosures at deeper levels don't restrict how fans can pair, so moving the level 1 conditional bracket over the fan-in can't affect how the fan is paired.

At this point, the different regions of the single conditional enclosure are touching and rule VII.c can merge them, giving (with the aid of another execution of rule II.a) graph H (The arc near the top of the graph indicates that everything below it is part of an enclosure). Rule VII.c required the conditional brackets to have the same level number; otherwise they would have belonged to different enclosures and should have passed through each other, following rule VII.n. What started in graph D as an enclosure with two disconnected regions has become a single large region. What started back in graph B as independent enclosures have become nested enclosures,

preparing the way for lambda reduction while avoiding possible mispairings of fans. The conditional brackets were the instrument of the transformation. In general, conditional enclosure boundaries merge disjoint regions by expanding them until they encounter an enclosing boundary; in this example there was no enclosing boundary so the conditional boundaries expanded to fill the entire surrounding graph structure.

Rule II.c could have been executed any time since graph D. Executing it now gives graph I, which is ready for rule I.b, a version of the beta reduction rule that accomodates a $\Cap_0$ node, giving graph J; the $\uplus_0$ keeps the λx node out of the enclosure. Rule IV.e duplicates the application to give graph K, where a fan-in is facing a fan-out across an enclosure boundary. The enclosure indicates that the fans are not paired, so they should duplicate each other, rather than connect corresponding links. Rule VI.a takes the first step, moving the fan-out inside the enclosure, but incrementing its number to give graph L. Just as with brackets, the number i on the fan indicates that the fan logically belongs i levels of enclosure out. Since the fans now facing each other have different numbers, they belong to different nestings of enclosures, and are thus not paired with each other; they should duplicate each other. All of the enclosure mechanism is ultimately in service of making sure that fans have the right numbers when they meet. Rule V.a does the actual duplication, yielding graph M, then two applications of rule VI.c take the fan-outs back out of the enclosure, yielding graph N. Even though rule VI.c is written upside down compared to the segments it matches in graph M, the rule fires; the graph is undirected, so rules can match in any orientation to the graph as a whole.

Only one other notable thing happens in the execution past graph N. Rule IV.a applies at graph N to move the $\uplus_0$ over the application. This splits the enclosure into two independent enclosures, which would be incorrect if fans in the two enclosures had interacted or if disconnected regions within the two enclosures has been part of one enclosure. This can't happen because a $\uplus$, called a restricted bracket, is never allowed to be used to bracket a fan from the point or to bracket a conditional bracket from the outside. These restrictions mean that an enclosure can always be split in the vicinity of a restricted bracket, jusifying the use of rule IV.a. The general bracket, $\cup$ doesn't have these restrictions, and so the algorithm knows less when it encounters one. For this reason, general brackets are used as little as possible. In fact, they only occur like $\cup_0$, as close brackets with number 0.

Outline of Proofs

Correctness

First, a quick summary of the algorithm: A legal graph is a rooted undirected graph and contains nodes of only the following types and arities. The ordinary nodes are the lambda nodes, variable nodes, and application nodes:

The control nodes are the fan nodes, general bracket nodes, restricted bracket nodes, and conditional bracket nodes:

As shown, each control node has an associated level number, a non-negative integer, which will always be 0 for general bracket nodes. Finally, there are two periphery node types, the root node and void nodes:

Each graph has one root node, which is its root; each graph presented so far should have had a root node connected to its top node. Void nodes haven't appeared in the examples; they connect to unused parts of a graph. Each arc of a node attaches at a distinguishable site, indicated in the pictures by relative location. In terms of standard graph theory, the nodes are labeled (the label is the node type, including the level number on a control node) and the arcs of a node are ordered.

Running the algorithm consists of encoding of a lambda expression into a graph, which will be described shortly, followed by executing the rewrite rules (then, optionally, reinterpreting the graph as a lambda expression). The rules of figure 3 are the heart of the algorithm. Rules I.a and I.b simulate beta reduction while the other rules of figure 3 get the control nodes out of the way of a potential beta reduction. Only the beta reduction rules change which lambda expression is represented by the graph. The rules in figure 4 eliminate unreachable graph structures (these result from reducing lambdas that have no instance of their bound variable). Those of figure 5 take advantage of special situations to simplify the graph structure, usually getting rid of a control node or two. The algorithm works without the rules of figure 5, but does less bookkeeping work if the rules are available. For example, when an implementation of the algorithm is run on a computation of 6! in the unary representation of Church numerals, it performs only about half as many

control rule executions when those rules are available and are given priority. All of the rules that move control nodes move them in a consistent direction; the algorithm can't get into a loop of just shuffling control nodes back and forth. Further, the rules are capable of getting any combination of control nodes out of the way of a potential beta reduction; if a lambda expression has a redex, the algorithm will be able to reduce it. It remains to show that the algorithm correctly simulates beta reduction.

The proof of correctness proceeds by defining the lambda expression represented by a graph and showing that an execution of the beta reduction rule on the graph does, in fact, simulate a collection of beta reductions on the lambda expression, while the remaining rules don't affect the represented lambda expression.

As described earlier, each proper path through the graph corresponds to a path through the lambda expression. A proper path is one which starts at the root and at each fan-out takes the branch corresponding to the branch it took into the paired fan-in. This can be defined precisely by defining the context accumulated by a path and how fan-outs find their matched fan-in in that context. For this discussion, we will use the informal description given in the previous section.

The definition of the tree represented by a graph glossed over the issue of how variables in the represented lambda expression should be named. Equivalently, it ignored the issue of how the scoping of lambdas is represented. Not surprisingly, the control nodes hold the answer. We present some properties relating lambda nodes to control nodes, which legal graphs must obey, show how to establish them in the initial encoding of the lambda expression as a graph, and show how they define how the graph represents scoping. The first three properties have already been mentioned in the examples.

There is a potential for confusion in the following definitions because the names "lambda" and "variable" might refer either to nodes in the graph or to vertices of the represented tree. To avoid ambiguity, we will always use *lambda node* and *variable node* to refer to nodes in the graph and use *lambda* and *variable occurrence* to refer to vertices in the tree.

Property 1 (pairing) *There is a one-to-one pairing between lambda nodes and variable nodes; a variable node represents all variable occurrences bound by all lambdas represented by its paired lambda node and represents nothing else.*

Since one lambda node can represent several lambdas, this property still allows one variable node to represent variable occurrences bound by several lambdas, provided its paired lambda node represents all those lambdas.

The property poses three minor difficulties in translating a lambda expression into a graph. First, it requires every variable to be bound by some lambda; this can be resolved by adding lambdas to the front of the expression to bind any otherwise unbound variables. Second, it presents a problem with expressions where a lambda doesn't bind any variables, for example $(\lambda x.(\lambda y.x))$. The solution is to go ahead and include variable nodes for all lambda nodes, but to connect unused variable nodes to void nodes. The example would be represented as

This situation is the way that void nodes are introduced into the graph. Later, if the λy is reduced, the argument will be connected to the $\otimes$node and garbage collected by the rules of figure 4. The final problem is that some lambda might bind several variable occurrences; in this case fan-ins are used to collect the references and connect them to the single variable node, as was done in the example.

Property 2 (transparency) *For any proper path segment that represents a sequence of links in the tree from a lambda to a variable occurrence bound by the lambda, the control nodes along the segment will have no net effect on any allowable context for the segment.*

That is, the segment yields the identity transformation on any allowable context. This property can be set up initially by using brackets to encapsulate any fan-ins that were introduced to satisfy the paring property, putting restricted brackets on each branch and general brackets on the points, as was done in the example.

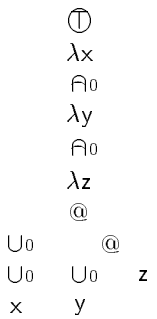
Property 3 (independence) *For any proper path to a lambda node, the make-up of the top level of its context at the lambda node will have no effect on how the path can be extended.*

More operationally, if control nodes that affected only the top level of the context were to be added just above the lambda node, they would not affect the tree represented by the graph. This is the property that justifies rule III.a, converting a general bracket above a lambda node to a conditional bracket.

Property 4 (nesting) *For any proper path segment that represents a sequence of links in the tree from a lambda to a variable occurrence free in the lambda, the context transformation determined by the segment will discard the make-up of the top level of the entering context.*

More operationally, if control nodes that affected only the top level of the context were to be added just above the lambda node, their effects would be discarded by any extension of the path that represents a variable occurrence free in the lambda. This property ensures that beta substitutions preserve the independence property; if an expression is substituted for free a variable inside a lambda, the top level of the context at the lambda won't be visible inside the expression. This property didn't come up in the example, because the example didn't have any free variables inside lambdas.

When initially encoding a lambda expression, this property is established by placing an open restricted bracket just above any lambda that contains free variables and placing one general close bracket just above a variable for each lambda in which the variable is free. This also ends up pairing up the open and close brackets, so as to preserve the transparency property. For example, the expression (λx. (λy. (λz. (x (y z))))) is represented by the graph



All four properties talk about which lambdas bind which variable occurrences, one in terms of the pairing of lambda nodes with variable nodes and the other three in terms of contexts. In a legal graph, they must agree. The transparency nesting properties are enough to completely determine which lambdas in the tree bind which variable occurrences. Even in a case where a path loops through the same lambda node several times before reaching a bound variable node, they determine which trip through the lambda node represents the lambda that binds the variable occurrence.

To see how the properties determine binding patterns, consider a path to a variable node and the variable occurrence it represents. Of the trips the path makes through lambda nodes, consider the latest one for which the make-up of its top level of context is still available at the variable node. The nesting property implies that the variable occurrence is not free in the lambda represented by that trip, so that lambda or some deeper one must bind the variable occurrence. But all deeper lambdas are represented by later trips through lambda nodes, for which the top level of the context is not available at the variable node. The transparency property implies that lambdas represented by those trips can't bind the variable, so the lambda in question must. In summary, for any path terminating at a variable node, the properties imply that the variable occurrence represented by the path is bound by the lambda represented by the nearest trip through a lambda node for which the make-up of the top level of the accumulated context is still available at the variable node.

We define the names of variables in the tree represented by the graph in accord with the properties. Every lambda in the tree is defined to bind a variable of a different name, and the name of each variable occurrence is then defined to be the same as that of that of lambda that the properties indicate should bind it. This definition assigns all tracking of scoping to the control nodes in the graph. Variable names only appear in the interpretation of the algorithm, not in the algorithm itself. For example, although the definition of beta reduction in the lambda calculus may require substantial renaming to avoid variable capture, the beta reduction rule in the algorithm only needs to change a few links; only the interpretation does renaming. Of course, the algorithm must end up with the same bindings that the lambda calculus would, and it does.

The proof of correctness is an induction, showing that if the graph satisfies the properties, then each rule preserves the properties and each rule preserves the lambda expression represented by the graph, except for the beta substitution rules (I.a and I.b), which simulate beta reductions.

Optimality

The optimality of the algorithm follows from its not duplicating nodes, except when necessary. In particular, application nodes are only duplicated if a fan-out is on their function link and lambda nodes are only duplicated if a fan-in is above them. These are exactly the two situations when a fan node is impeding a potential beta reduction.

The optimality is demonstrated by relating the algorithm's copying with a labelling on lambda expression links that Lévy's defines to specify optimality. Lèvy defines a parallel reduction step consisting of reducing all identically labelled redexes in one step. The task is to show that each beta reduction step of the algorithm corresponds to one of Lèvy's parallel reductions.

It would be nice if any two links in a lambda expression with identical labels under Lévy's labelling were represented by a single link in the graph, but the actual case is slightly more involved. It is possible to define the the *prerequisite chain* of a vertex in the lambda expression, which will be a sequence of links in the lambda expression, all of which must be involved in beta reductions before the vertex can be involved in a beta reduction. Then we

can show

Lemma 1 *At any point during the execution of the rules of the algorithm, if two vertices in the tree represented by the graph have prerequisite chains of the same length and if corresponding links in the chains have matching labels, then the chains have the same representation in the graph.*

In the case of redexes, the link from the application to the function is a prerequsite chain by itself, and so the lemma guarantees that all identically labelled redexes will be represented by the same structure in the graph, and thus reduced in a single step.

Lèvy's results then imply that no order of executing the rules of the algorithm will duplicate work. But it is possible that some rule executions might do useless work, that is do beta reductions inside a subexpression that is eventually discarded. The solution is to impose a normal order strategy on the rule execution. An implementation would keep track of part of the graph represents the redex that would be reduced in normal order (this can be done efficiently) and would only execute rules on that part.

Of course, such a strategy reduces the opportunities for parallelism. A parallel implementation of the rules might dispense with normal order to achieve high parallelism at the cost of possibly doing some work that was later discarded. The proof that the rules never duplicate work would still apply to such a system.

Acknowledgments

Jim desRivieres and Jean-Jacques Lévy helped simplify the algorithm. They and Alan Bawden, Pavel Curtis, Dan Friedman, Julia Lawall, and Dan Rabin suggested substantial improvements to earlier drafts.

References

[1] Barendregt, H. P. *The Lambda Calculus : its Syntax and Semantics*, vol. 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1981.

[2] Barendregt, H. P., et al. LEAN, an intermediate language based on graph rewriting. *Parallel Computing 9* (1989), 163–177.

[3] Kathail, V. private communication, 1989.

[4] Lévy, J.-J. *Réductions correctes et optimales dans le lambda-calcul.* PhD thesis, Université de Paris, 1978.

[5] Lévy, J.-J. Optimal reductions in the lambda-calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Seldin and J. Hindley, Eds. Academic Press, 1980, pp. 159–191.

[6] Staples, J. Efficient evaluation of lambda expressions: a new strategy. Tech. Rep. 23, University of Queensland, Department of Computer Science, St. Lucia, Queensland, 4067, Australia, 1980.

[7] Staples, J. Two-level expression representation for faster evaluation. In *Graph-Grammars and their Application to Computer Science: 2nd International Workshop*, H. Ehrig, M. Nagl, and G. Rozenberg, Eds. Springer-Verlag, 1982. Lecture Notes in Computer Science 153.

[8] Turner, D. A. A new implementation technique for applicative languages. *Software Practice and Experience 9*, 1 (1979).

[9] Wadsworth, C. P. *Semantics and Pragmatics of the λ-calculus.* PhD thesis, Oxford University , 1971.

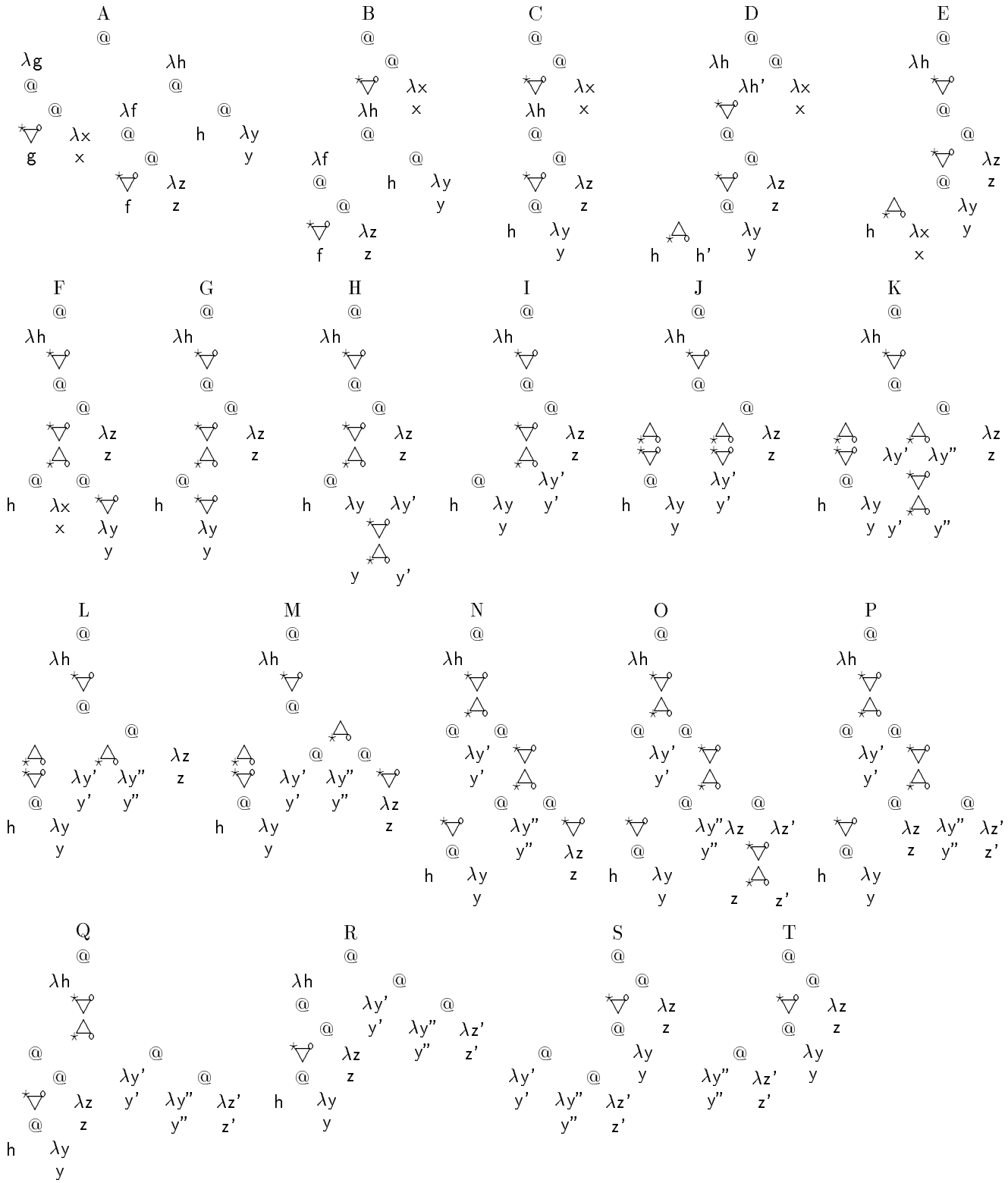


Figure 1: Simplified Execution

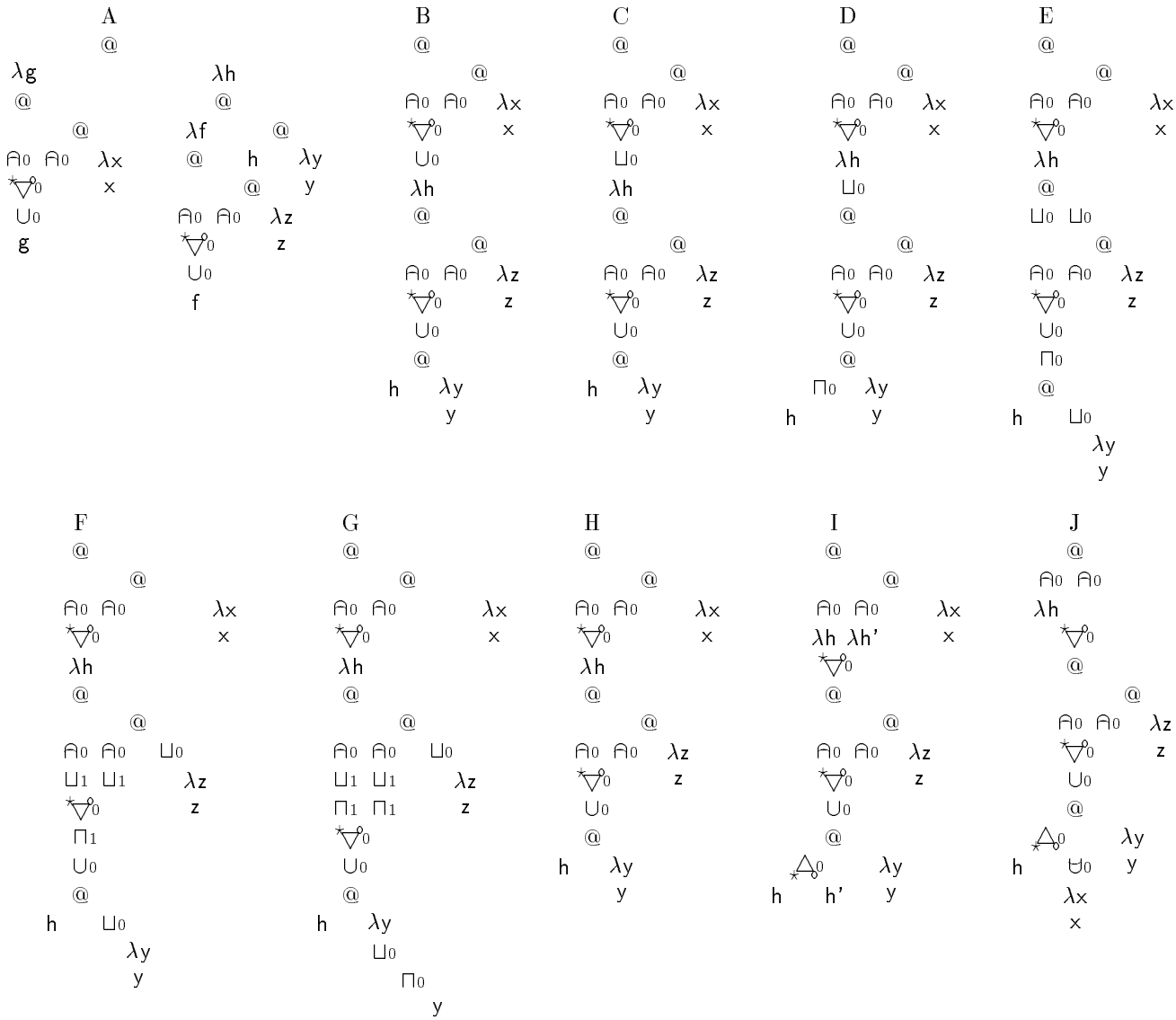
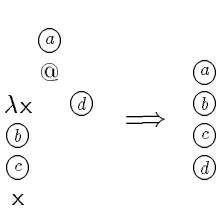
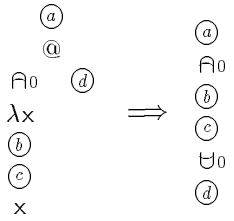

A
@
λg
@
@
Α0 Α0 λx
*▽0 x
U0
g
λh
@
λf @
@ h λy
@ y
Α0 Α0 λz
*▽0 z
U0
f
B
@
@
Α0 Α0 λx
*▽0 x
U0
λh
@
@
Α0 Α0 λz
*▽0 z
U0
@
h λy
y
C
@
@
Α0 Α0 λx
*▽0 x
Ш0
λh
@
@
Α0 Α0 λz
*▽0 z
U0
@
h λy
y
D
@
@
Α0 Α0 λx
*▽0 x
λh
Ш0
@
@
Α0 Α0 λz
*▽0 z
U0
@
Π0 λy
h y
E
@
@
Α0 Α0 λx
*▽0 x
λh
@
Ш0 Ш0
@
Α0 Α0 λz
*▽0 z
U0
Π0
@
h Ш0
λy
y
F
@
@
Α0 Α0 λx
*▽0 x
λh
@
@
Α0 Α0 Ш0
Ш1 Ш1 λz
*▽0 z
Π1
U0
@
h Ш0
λy
y
G
@
@
Α0 Α0 λx
*▽0 x
λh
@
@
Α0 Α0 Ш0
Ш1 Ш1 λz
Π1 Π1 z
*▽0
U0
@
h λy
Ш0
Π0
y
H
@
@
Α0 Α0 λx
*▽0 x
λh
@
@
Α0 Α0 λz
*▽0 z
U0
@
h λy
y
I
@
@
Α0 Α0 λx
λh λh' x
*▽0
@
@
Α0 Α0 λz
*▽0 z
U0
@
*△0 λy
h h' y
J
@
Α0 Α0
λh
*▽0
@
@
Α0 Α0 λz
*▽0 z
U0
@
*△0 λy
h Θ0 y
λx
x
K
@
Α0 Α0
λh
*▽0
@
@
Α0 Α0 λz
*▽0 z
U0
*△0
@ @
h *▽0
λy
Θ0 y
λx
x
L
@
Α0 Α0
λh
*▽0
@
@
Α0 Α0 λz
*▽0 z
*△1
U0 U0
@ @
h *▽0
λy
Θ0 y
λx
x
M
@
Α0 Α0
λh
*▽0
@
@
Α0 Α0 λz
*△1 *△1 z
*▽0 *▽0
U0 U0
@ @
h *▽0
λy
Θ0 y
λx
x
N
@
Α0 Α0
λh
*▽0
@
@
*△0 *△0 λz
Α0 Α0 Α0 Α0 z
*▽0 *▽0
U0 U0
@ @
h *▽0
λy
Θ0 y
λx
x

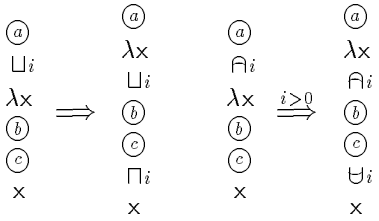
Figure 2: Illustrated Execution



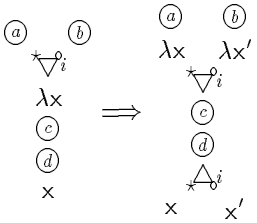
I.a



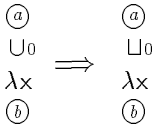
I.b



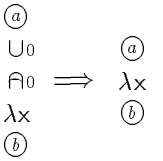
II.a II.b



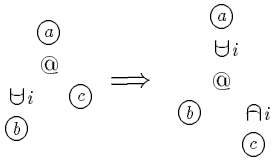
II.c



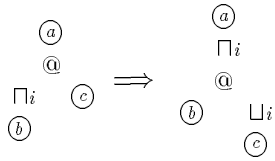
III.a



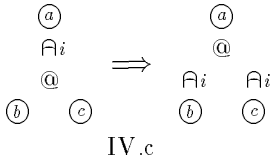
III.b



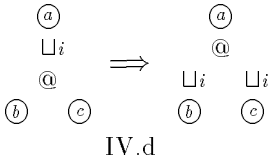
IV.a



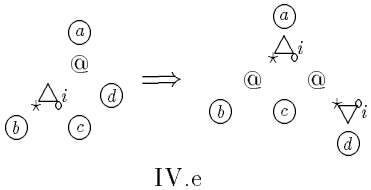
IV.b



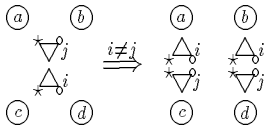
IV.c



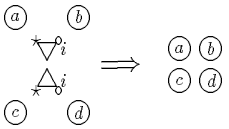
IV.d



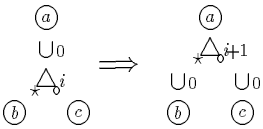
IV.e



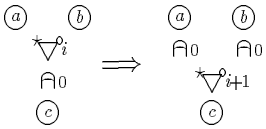
V.a



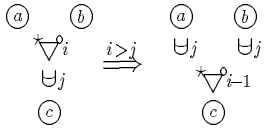
V.b



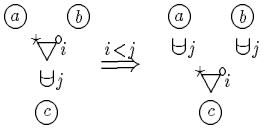
VI.a



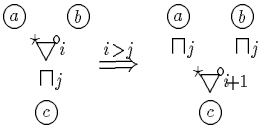
VI.b



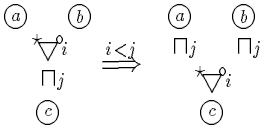
VI.c



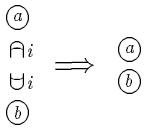
VI.d



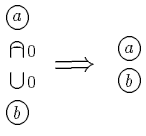
VI.e



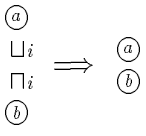
VI.f



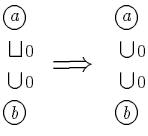
VII.a



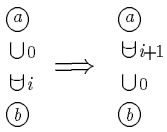
VII.b



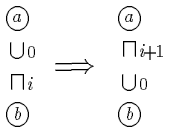
VII.c



VII.d



VII.e



VII.f