Parallel standard translation between Lambda Calculus and Combinatory Logic

Agnieszka Lupinska
agnieszka.lupinska [at] uj.edu.pl
Theoretical Computer Science
Jagiellonian University

Simulation λ-abstraction by **K** and **S**

Definition

We define $\lambda^* x.P$ by induction on the structure P as follows:

$$\lambda^* x.x \equiv I$$
$$\lambda^* x.P \equiv \mathbf{K}P \text{ if } x \notin FV(P)$$
$$\lambda^* x.PQ \equiv \mathbf{S}(\lambda^* x.P)(\lambda^* x.Q)$$

Notation

Let $\vec{x} \equiv x_1, \ldots x_n$.
Then $\lambda^* x_1 x_2 \ldots x_n.T \equiv \lambda^* \vec{x}.T \equiv \lambda^* x_1.\big(\lambda^* x_2.\big(\ldots\big(\lambda^* x_n.T\big)\ldots\big)\big)$

How it works

$\lambda^* xy.yx \equiv \lambda^* x.(\lambda^* y.yx) \equiv \lambda^* x.\mathbf{S}(\lambda^* y.y)(\lambda^* y.x) \equiv \lambda^* x.\mathbf{SI}(\mathbf{K}x) \equiv$
$\mathbf{S}(\mathbf{K}(\mathbf{SI}))(\mathbf{S}(\mathbf{KK})\mathbf{I})$

Standard translation

Definition

Standard translation from Combinatory Logic to Lambda Calculus:

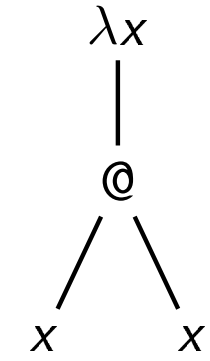
$$
\begin{aligned}
&[x]_\lambda \equiv x \\
&[\mathbf{K}]_\lambda \equiv \lambda xy.x \\
&[\mathbf{S}]_\lambda \equiv \lambda xyz.xz(yz) \\
&[PQ]_\lambda \equiv [P]_\lambda [Q]_\lambda
\end{aligned}
$$

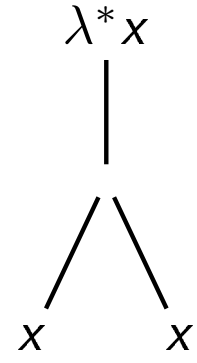
Standard translation from Lambda Calculus to Combinatory Logic:

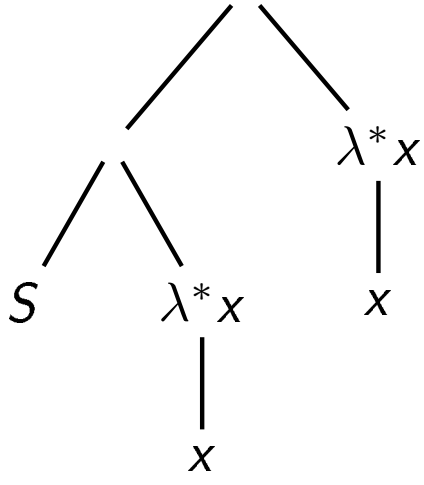
$$
\begin{aligned}
&[x]_{CL} \equiv x \\
&[MN]_{CL} \equiv [M]_{CL}[N]_{CL} \\
&[\lambda x.M]_{CL} \equiv \lambda^* x.[M]_{CL}
\end{aligned}
$$

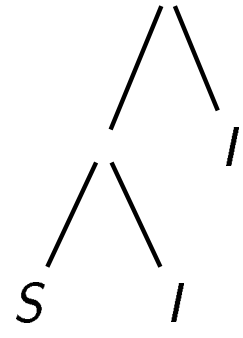
Example for $\lambda x.xx$

$$[\lambda x.xx]_{CL} \equiv \lambda^* x.[xx]_{CL} \equiv \lambda^* x.xx = S(\lambda^* x.x)(\lambda^* x.x) = SII$$

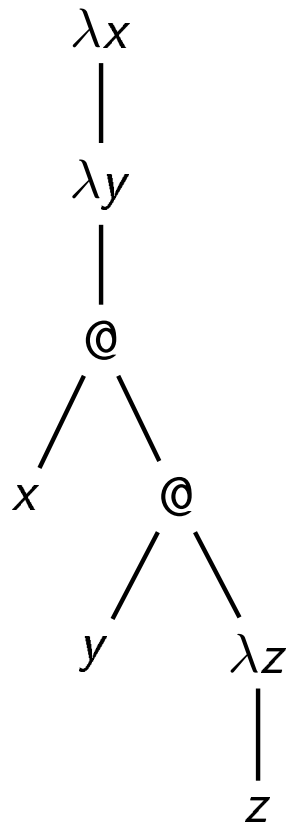








Example for $\lambda xy.x(y\lambda z.z)$



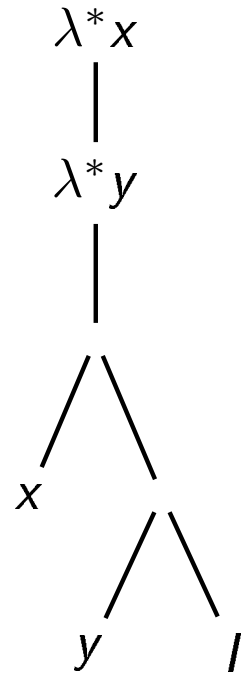
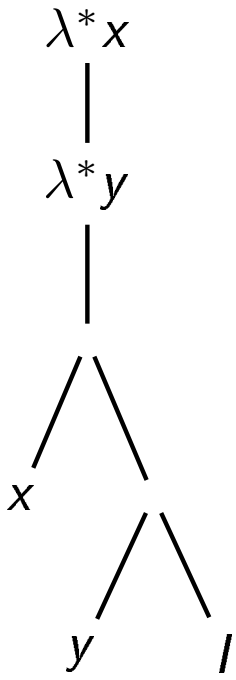


Figure: From definition of translation to CL, removing λ^* abstraction z

$[\lambda xy.x(y\lambda z.z)]_{CL} = \lambda^* xy.[x(y\lambda z.z)]_{CL} =$
$\lambda^* xy.[x]_{CL}[y\lambda z.z]_{CL} = \lambda^* xy.x([y]_{CL}[\lambda z.z]_{CL}) =$
$\lambda^* xy.x(y\lambda^* z.z) = \lambda^* xy.x(yI)$



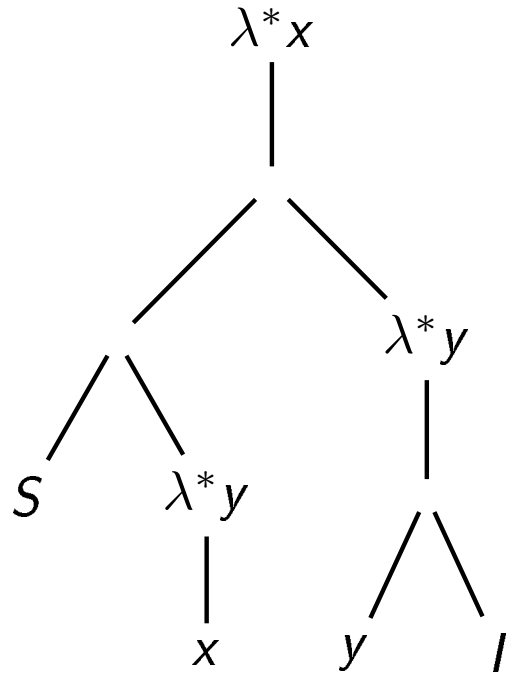
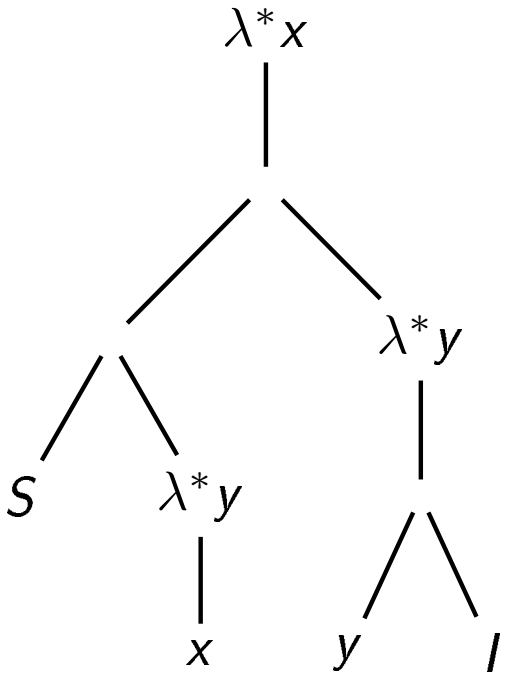


Figure: Start removing of $\lambda^* y$
$\lambda^* xy.x(yI) = \lambda^* x.\lambda^* y.x(yI) = \lambda^* x.S(\lambda^* y.x)(\lambda^* y.yI)$



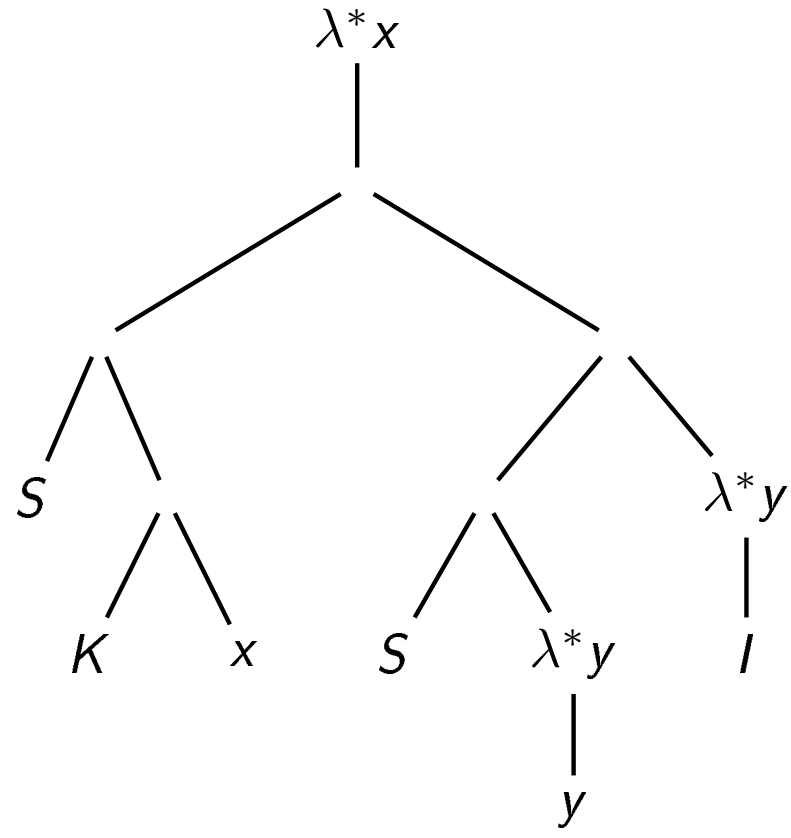
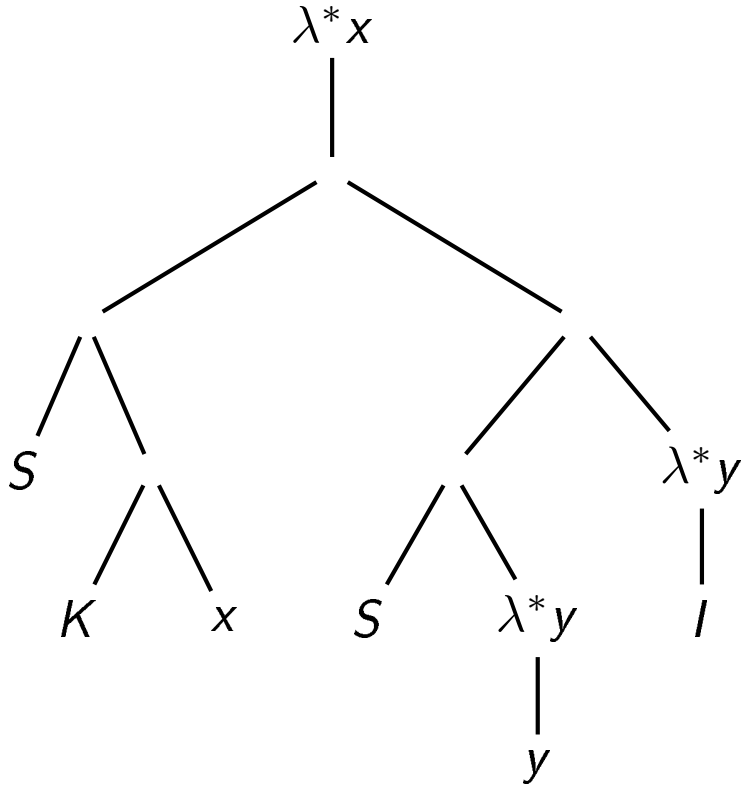


Figure: $\lambda^* x.S(\lambda^* y.x)(\lambda^* y.yI) = \lambda^* x.S(Kx)(S(\lambda^* y.y)(\lambda^* y.I))$



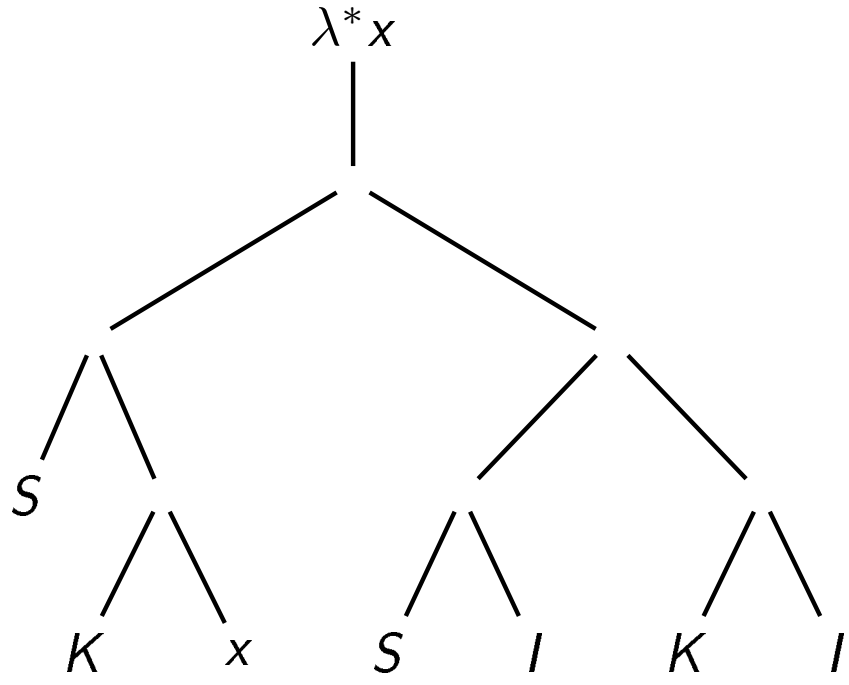
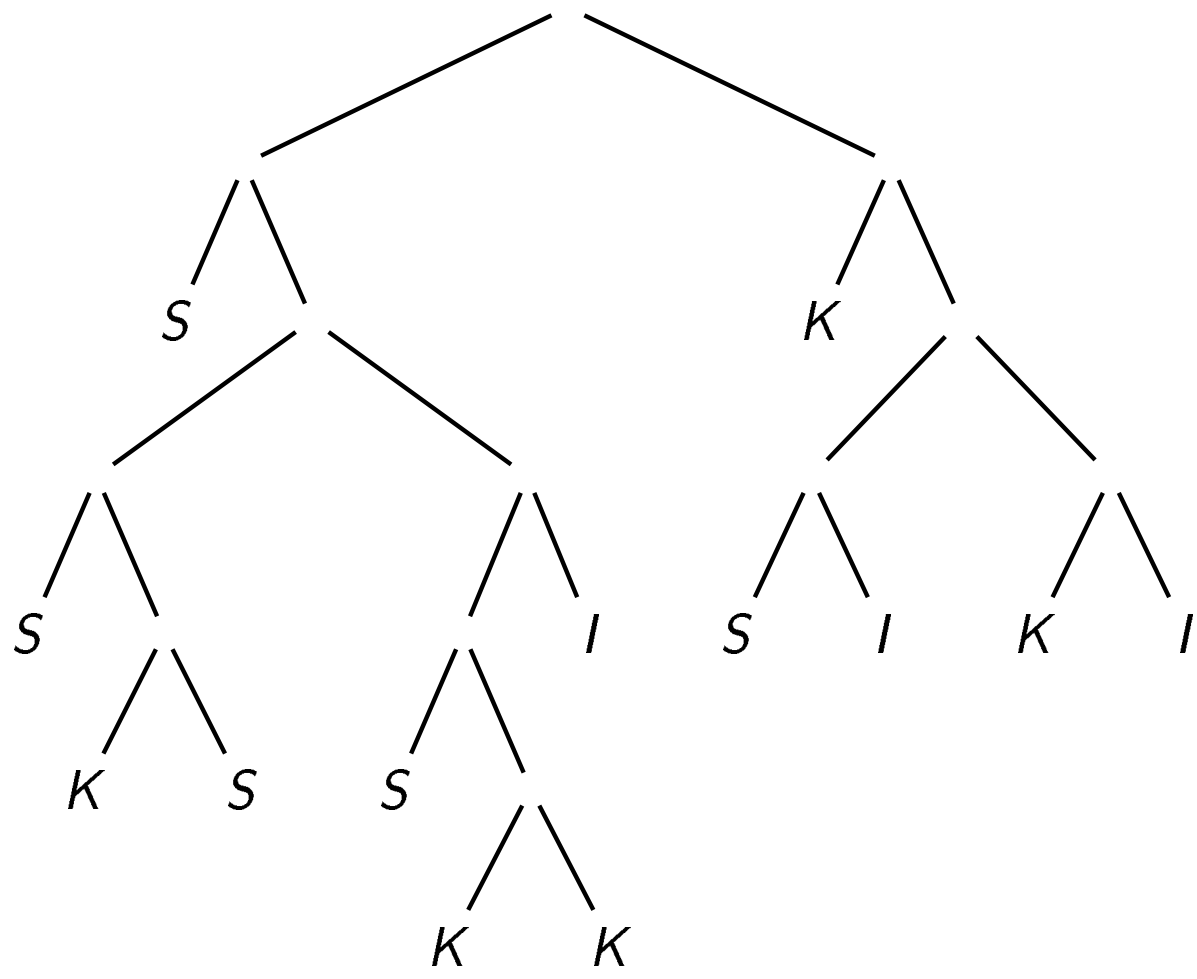


Figure: After removing the second λ^* abstraction y
$\lambda^* x.S(Kx)(S(\lambda^* y.y)(\lambda^* y.I)) =$
$\lambda^* x.S(Kx)(SI(KI))$



After removin $\lambda^* x$ we have got
$\lambda xy.x(y\lambda z.z) = S(S(KS)(S(KK)I))(K(SI(KI)))$

1. For each $\lambda^* x$-abstraction a tree is rebuild only in nodes which lie on the paths leading from leaves x to the root.
2. Each λ^*-abstraction is removing separately of others.

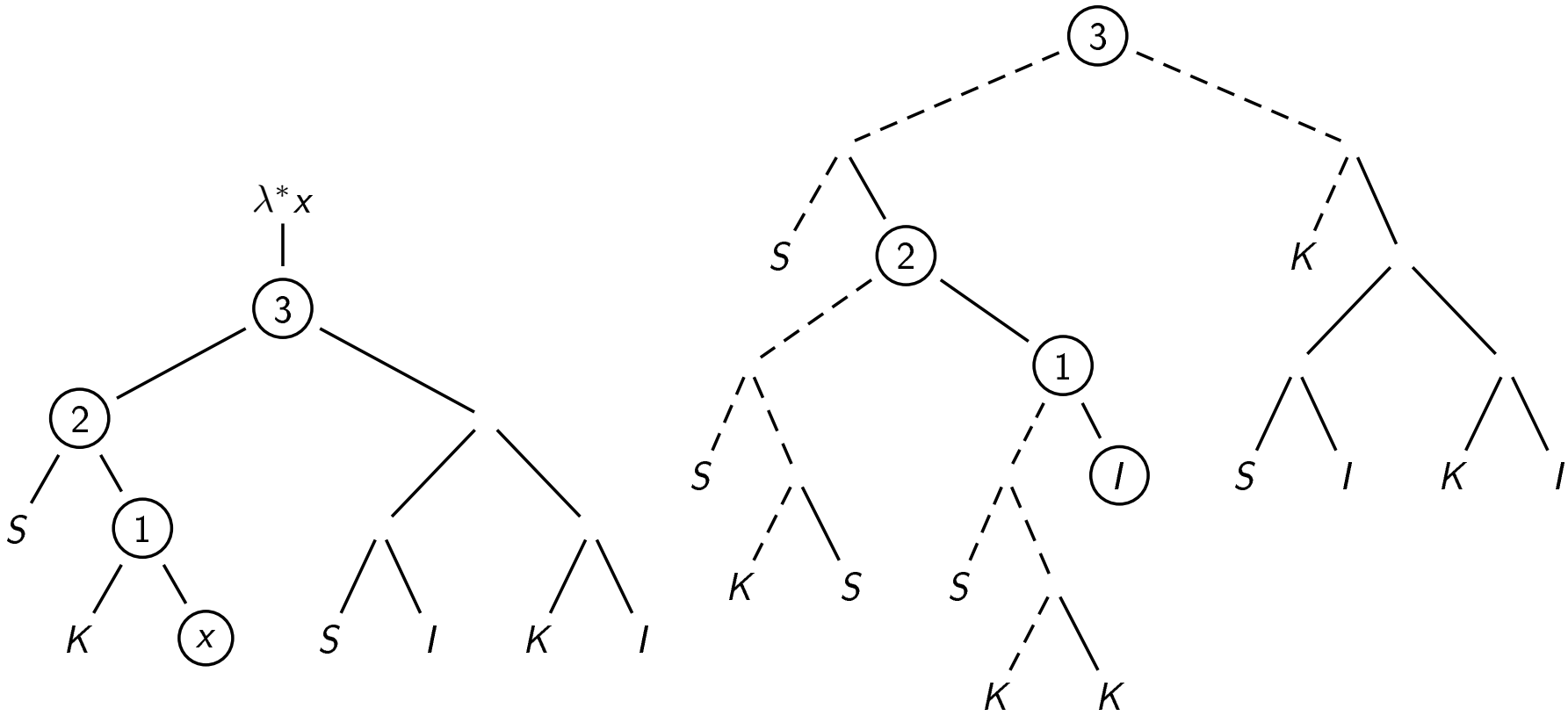


Figure: After removing $\lambda^* x$

Active node

Definition

Let x be a variable in a combinator C and let $\lambda^* x$ be an abstraction removing from C during the ST algorithm. A node is an ***active*** *if it is on a path leading from a leaf x to the root.*

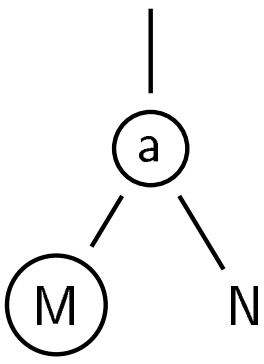
We show that for each λ^*-abstraction, all operations performed on active nodes are independently to each other. Let a be an active node in a tree

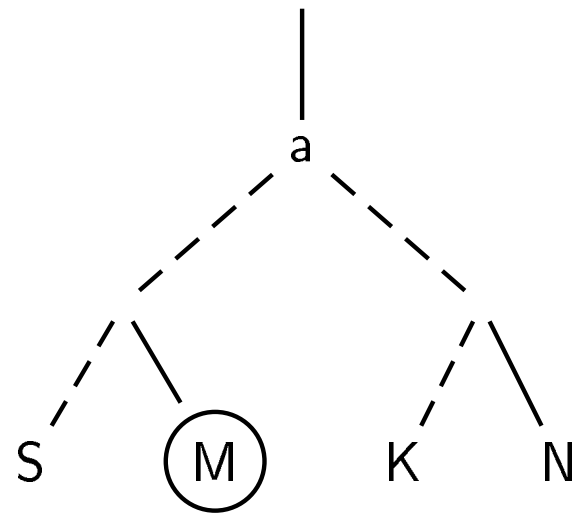
of some combinator. We show that for each active node we have to add at most 4 new nodes and the places of the new nodes in a tree are determined. We are given the following cases.

The active node a is a leaf.

Since we allow the abbreviation: $I = SKK$ then the node a is changed to combinator I. There are no new nodes.

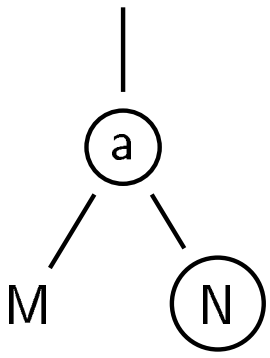
Only the left child of a is active.

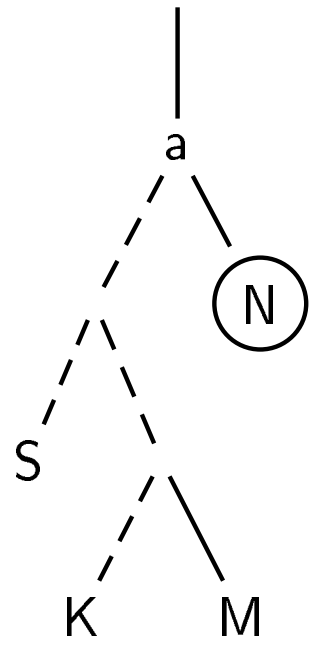




There are 4 new nodes in the tree: S, K and two internal nodes. Notice the node a does not change its pointer to the father in the tree. The node a changes the pointer to the father of the active node M.

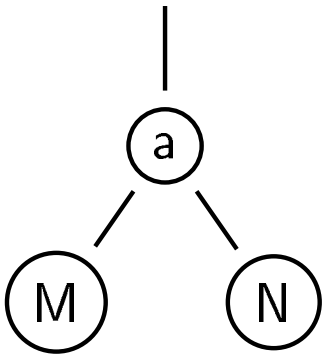
Only the right child of a is active.

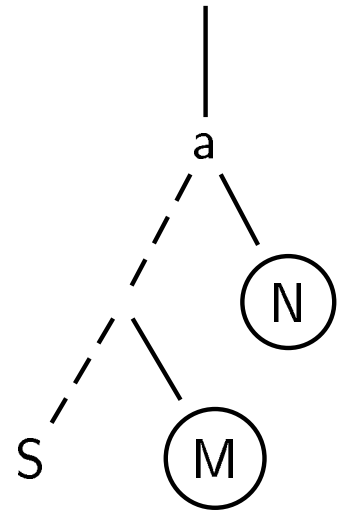




There are 4 new nodes in the tree: S, K and two internal nodes. Notice the node a does not change its pointer to the father in the tree. The node a changes the pointer to the father of the active node N.

The left and the right child of n are active.





There are 2 new nodes in the tree: S and one internal node. Notice the node a does not change its pointer to the father in the tree. The node a changes the pointer to the father of the active node M.

Since there are no more cases, we see that for each λ^*-abstraction and for each active node the algorithm adds at most 4 new nodes to the combinator tree and the places of the new nodes are determined and independent from new nodes added by other active nodes.

Moreover, in each case an active node do not change its pointer to the father. This pointer can be changed only by its father. Therefore the ST algorithm can be performe in parallel.

In our approach all λ^*-abstractions are performed in order, from the deepest towards the root. For each λ^*-abstraction the active nodes are computed in parallel. Then for all active nodes the tree is rebuilded in parallel.

Parallel ST Algorithm

Algorithm 1 The parallel standard translation algorithm

```
1:  function STranslation(x)
2:      if x is leaf then
3:          Do nothing
4:      else
5:          if x is internal then
6:              STranslation(left(x))
7:              STranslation(right(x))
8:          else
9:              // x is λ∗ abstraction
10:             STranslation(left(x))
11:             Mark in parallel all active nodes
12:             For all active nodes rebuild tree in parallel
13:         end if
14:     end if
15: end function
```

Active nodes

We use N threads assigned to N nodes in a tree. Let $\lambda^* x$ be an abstraction reduced during the algorithm. The computition of active nodes can be perfomed in two different ways to execute time.

The time of the first method is linear due to the size of a tree. We choose the threads assigned to the leaves. If the leaf is a variable x then a thread goes towards the λ^*-abstraction and marks all nodes on the path as active.

In the second method the execution time is constant but algorithm needs the additional memory of quadratic size due to the size of a tree. Let x be a node in a tree. The *active* array is computed at the begining of the algorithm. Then the array is updated during adding new nodes.

Algorithm 2 The active array computation

```
 1: function MarkActive(x)
 2:     if left(x) = null and right(x) = null then
 3:         // x is leaf
 4:         active[x] ← {x}
 5:     else
 6:         if left(x) ≠ null and right(x) ≠ null then
 7:             // x is internal
 8:             MarkActive(left(x))
 9:             MarkActive(right(x))
10:             active[x] ← active[left(x)] ∪ active[right(x)]
11:         else
12:             // x is λ* abstraction
13:             MarkActive(left(x))
14:             active[x] ← active[left(x)] - {x}
15:         end if
16:     end if
17: end function
```

Algorithm 3 Rebuild tree

```
1: function RebuildTree(x)
2:     if x is active then
3:         if left(x) = null and right(x) = null then
4:             x ← I
5:         else
6:             if left(x) is active and right(x) is active then
7:                 Add 2 new nodes and set up active nodes
8:             else
9:                 if left(x) is active then
10:                    Add 4 new nodes and set up active nodes
11:                else
12:                    Add 4 new nodes and set up active nodes
13:                end if
14:            end if
15:        end if
16:    end if
17: end function
```

Hendrik Pieter Barendregt, 1984. *The Lambda Calculus, Its Syntax and Semantics* Studies in Logic and the Foundations of Mathematics, Volume 103, North-Holland. ISBN 0-444-87508-5