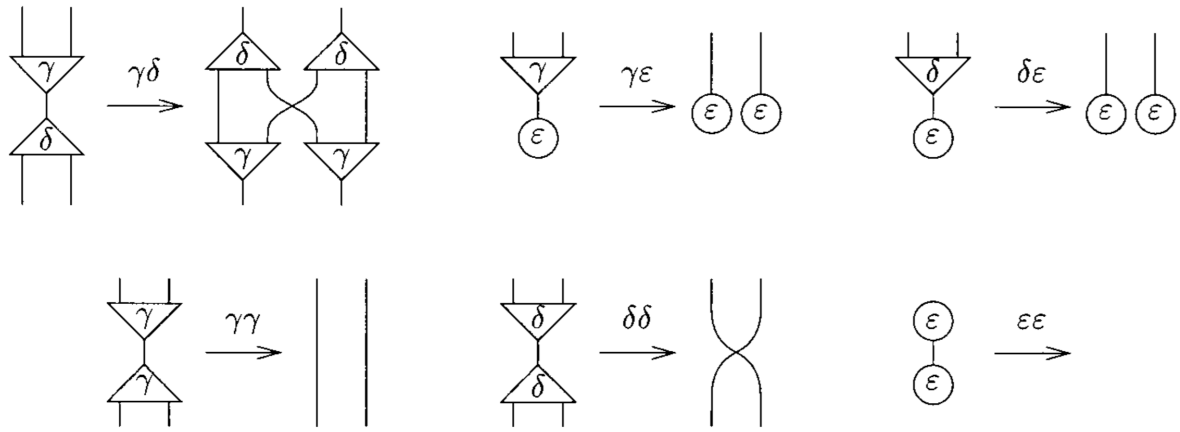
Simulating the λ-Calculus with Interaction Combinators

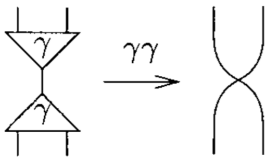
Nadav Kohen

1 Introduction

LaFont [1] defined a system of three interaction combinators that together are Turing-complete. The three combinators are γ (constructor), δ (duplicator), and ϵ (eraser). Their interaction rules are defined in Figure 2 taken from LaFont's paper.



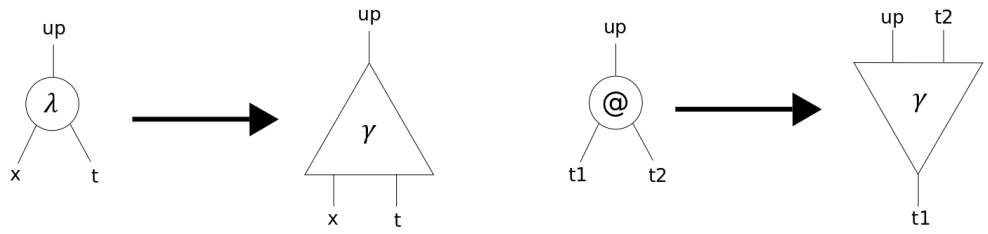
In this paper, we will replace the $\gamma\gamma$ rule with



since this will simplify our encoding of lambda terms into interaction combinators and make reduction code simpler as it can take advantage of the symmetry of γ and δ.

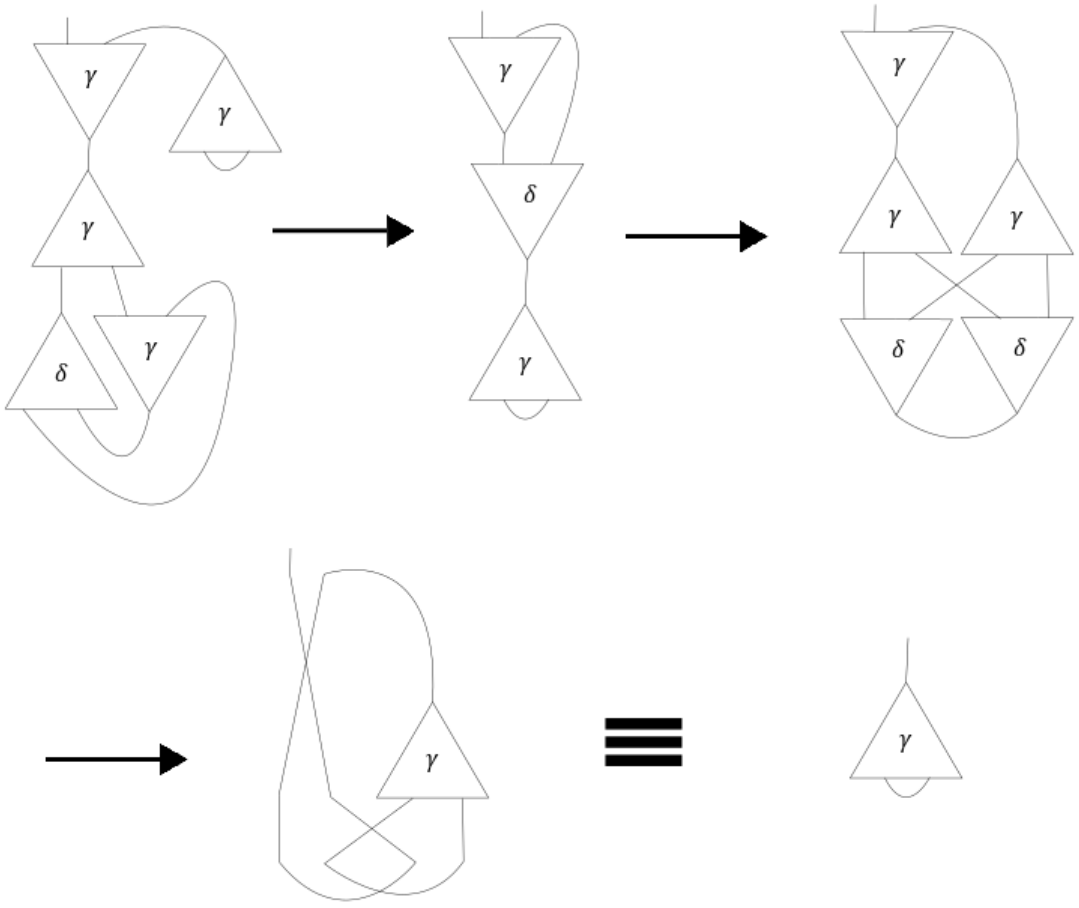
2 A First Approach

The idea behind this first approach is to view λ-terms as trees and encode them as interaction nets where every β-redex in the term becomes a cut in the circuit.
As such we encode λ-abstractions as upward facing constructors, and applications as downward facing constructors as follows:



We encode variables by wires leading to where the variable is bound or the free port corresponding to that free variable. In the case that a bound variable occurs multiple times, use a chain/tree of δ cells.

Example 1. Here is an example reduction, $(\lambda\ x\ .\ x\ x)(\lambda\ x\ .\ x) \rightsquigarrow_\beta^* \lambda\ x\ .\ x$



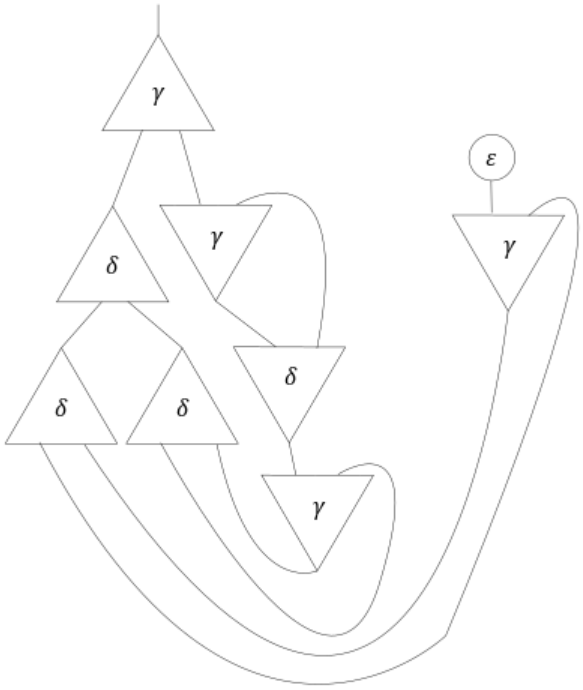
As it is, this system clearly reduces at least some terms successfully. But there are two issues that need to be addressed.

2.1 Duplicating and Deleting Applications

Because application γs have their principal ports pointing down (i.e. their parent in the tree is not attached to the principal port), a δ or ϵ is unable to duplicate or delete the sub-term in general. Note, that this is only an issue for sub-terms that are not closed (i.e. there are

free variables or variables bound outside of this sub-term) since deleting a closed term will send ϵs along all variable bindings which will reach the principal ports of applications, and the same goes for δ if the closed term is linear (more on that in the next section).

This issue has a relatively straightforward solution due to the fact that 'withholding' a sub-term with a top-level application during substitution will not hide any β-redexes. That is to say that even if an application γ is not duplicated or deleted and has a δ or ϵ waiting on its right port, all β-reductions will still have been performed (both within the sub-term and above). Hence, this problem can be solved during the translation of the normalized net back into a term by moving past any δ as the tree is being traversed. Furthermore, deletion isn't much of an issue since deleted parts cannot be reached from the root so that they will not appear in the resulting term. For example, the following net would be translated into the term $\lambda\ x\ .\ (x\ x)(x\ x)$.



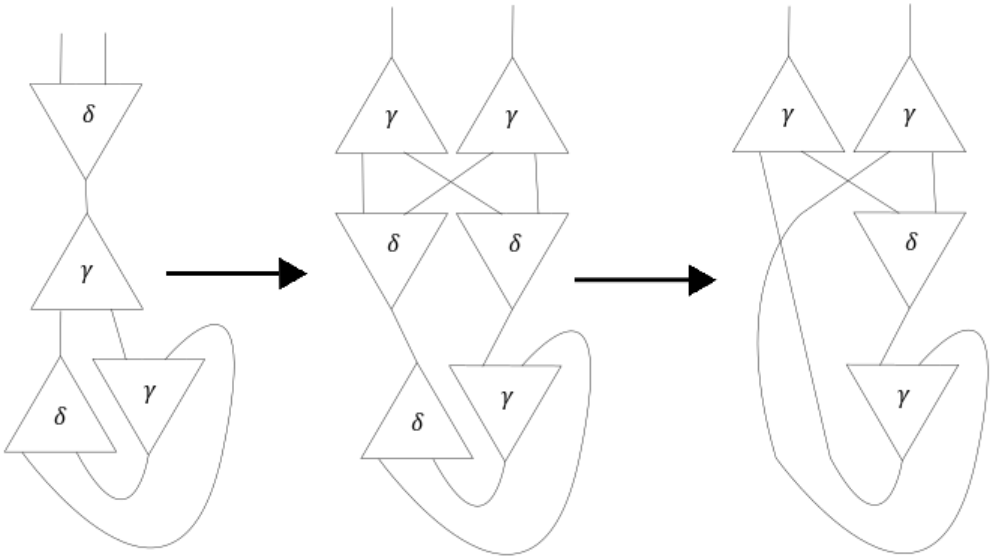
However, this solution has two drawbacks as it pertains to deletion: unnecessary work might be done underneath an ϵ which doesn't pass an application γ, and the more important problem that terms such as $(\lambda\ x\ .\ y)((\lambda\ x\ .\ x\ x)(\lambda\ x\ .\ x\ x))$ will not terminate.

One bad but easy to implement solution is to manually do garbage collection (i.e. remove 'deleted' wires from the queue of cuts waiting to be reduced) periodically after some fixed number of reduction steps.

A better solution would be to mark γ's that are lambdas with an ϵ to the left (unused bound variables), and if they eliminate with another γ (which would be an application), then send a signal through the sub-net to be deleted removing all wires from the queue of cuts waiting to be reduced.

2.2 Duplicating Non-Linear Terms

Since δ can only duplicate nets that contain no δ's (as shown in [1]), the current system cannot duplicate non-linear terms, such as $\lambda\ x\ .\ x\ x$



This means our system to be unable to reduce such terms as $(\lambda\ x\ .\ \lambda\ y\ .\ y\ x\ x)(\lambda\ x\ .\ x\ x)$.

This problem is not as easily fixed as the problem of duplicating and deleting applications. The remainder of this paper will explore three solutions to resolving this problem.

3 Full Parallel Reduction

One solution is to make a term into an interaction net, reduce all immediate cuts (those in the original net with no further reduction), translate back into a term, and then repeat. Note that translating the net into a term manually does duplication. Translating also does deletion since ϵ's will not be reached from the root so this implementation would not need to do any extra form of manual deletion like the methods at the end of section 2.1.

This solution takes advantage of the fact that given any lambda term encoded as an interaction net (as in section 2), if every cut in the original net is reduced, then the resulting net is a well-formed encoding of a lambda term (under the extended translation specified in section 2.1). Specifically this method corresponds to one full parallel reduction step.

This approach has the benefit that when a sub-term needs to be deleted, it cannot be reached from the root and is immediately discarded so that no further reduction is done beneath any ϵ (e.g. $(\lambda\ x\ .\ y)((\lambda\ x\ .\ x\ x)(\lambda\ x\ .\ x\ x))$ will normalize to y in a single step). In particular this ensures termination of normalizing terms.
A drawback of this approach is the large overhead of translating back and forth between terms and interaction nets between each parallel step, especially since this process is not very

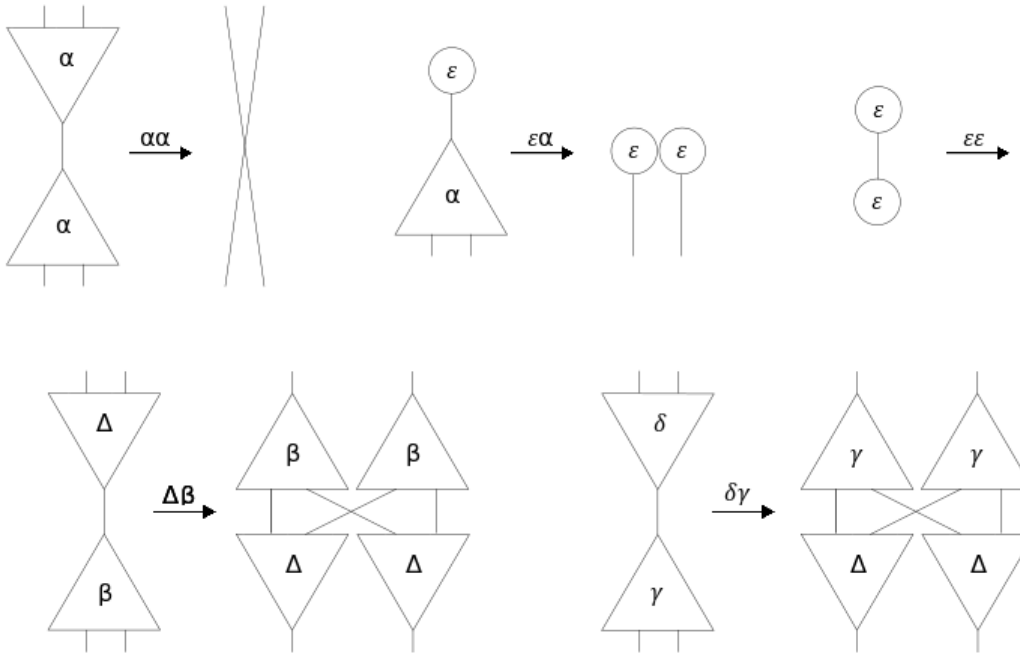
parallel. That being said, the actual reduction of the net can still be parallel in between traversals of the nets/trees.

4 Coding and Decoding Sub-Terms with Delta

Another solution to the problem posed by duplicating terms with δ in them is to use a different encoding of lambda terms that does not use any δ's underneath any lambda abstractions (since they only appear in our current encoding to the left of an abstraction). This approach is detailed in section 3 of Mackie and Pinto's paper [2]. The key idea is to use a construction from section 2.6 of LaFont's paper which he calls coding and decoding, although Mackie and Pinto call this packing and unpacking (in section 2.3 of [2]). Since this approach is fully described (and soundness proven) in their paper, I will move on to the next solution which is (to my knowledge) original.

5 A New System of Four Interaction Combinators to Efficiently Simulate Lambda Reduction

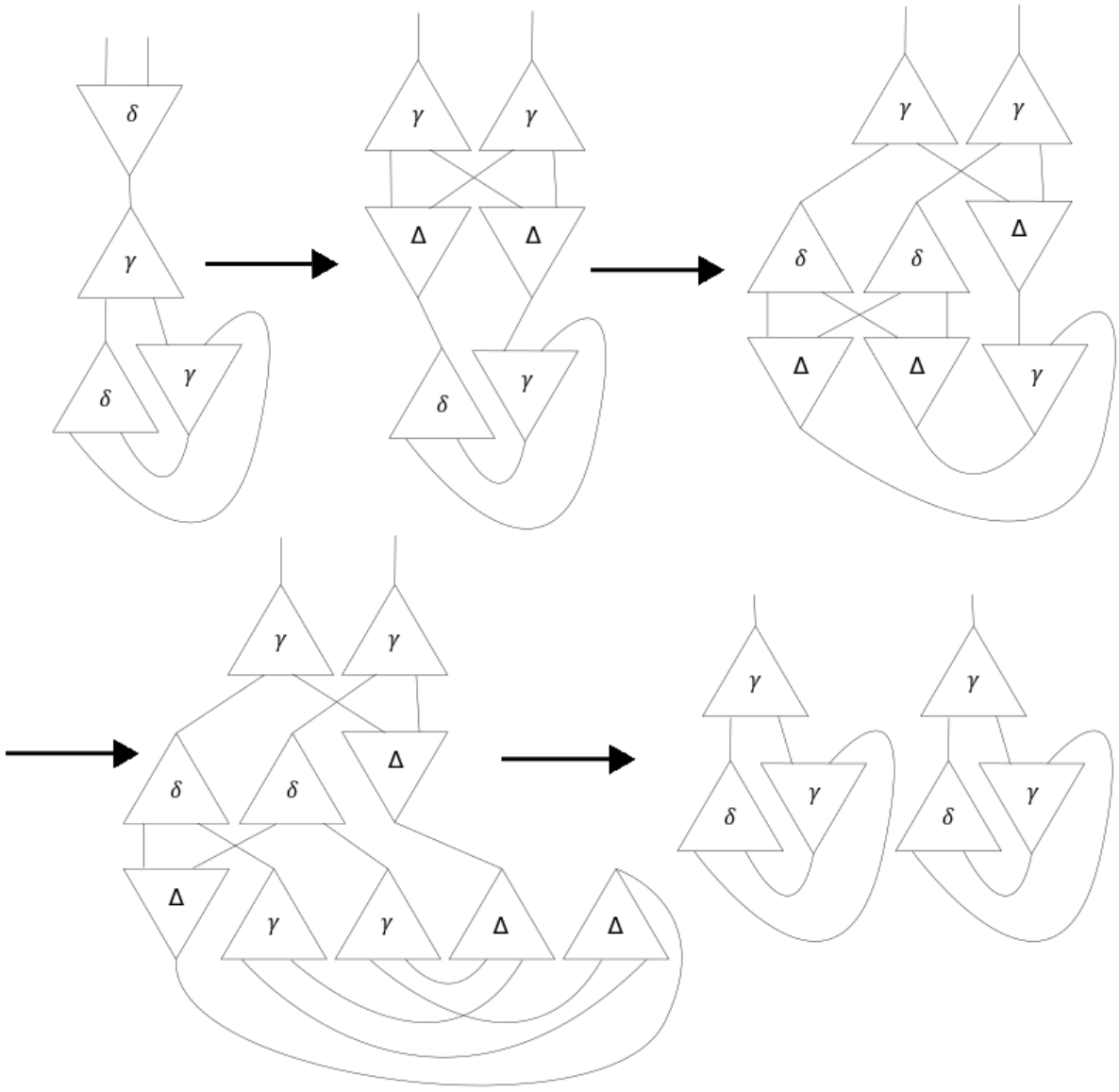
I believe the best way to solve the issue of duplicating duplicators is to add a new combinator to the system, the Duplication Constructor. This new system still has γ (constructor), ϵ (eraser), and Δ (duplicator), but now δ will be the duplication constructor. The following depicts the reduction rules where $\alpha \in \{\gamma, \delta, \Delta\}$ and $\beta \in \{\gamma, \delta\}$



We encode lambda terms with γ, δ, and ϵ just as in section 2 (where now δ is the duplication constructor not the duplicator). Now, during substitution, if a sub-term needs to be duplicated it will have a δ attached to its root and if the sub-term is a variable, then there is nothing to do, if it is an application, then there is also nothing to do since we still

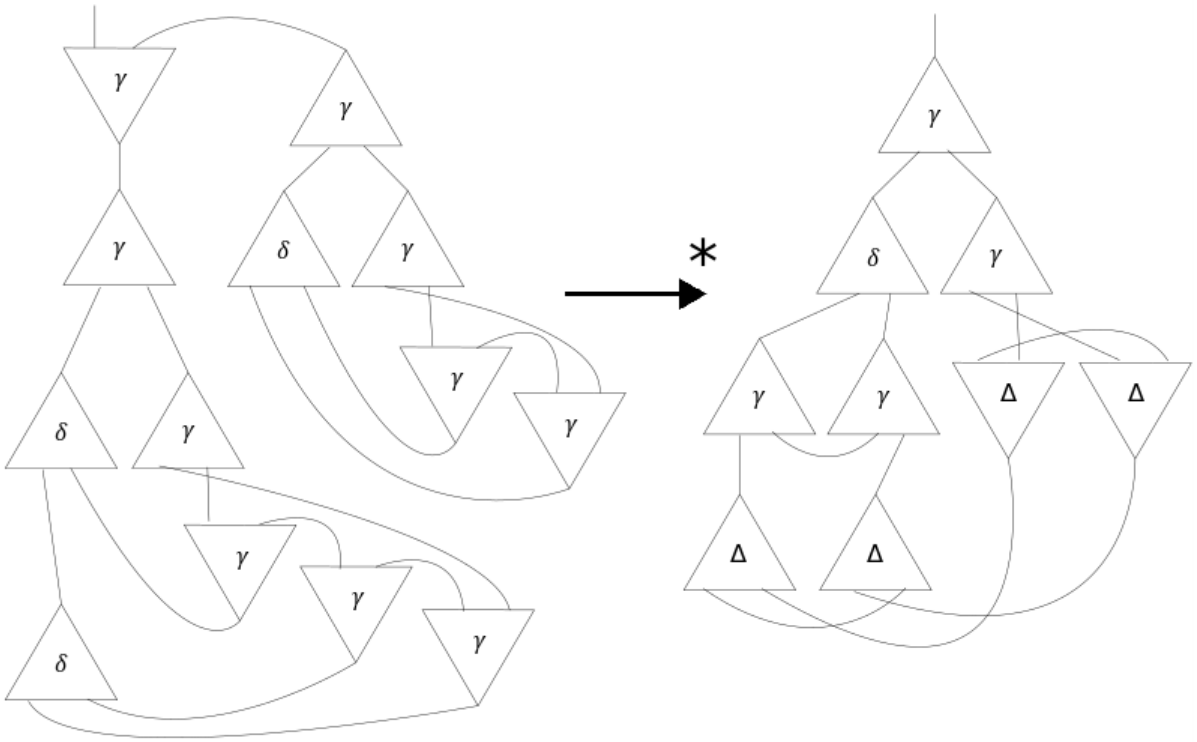
use the translation trick of moving past δ from section 2.1, and lastly if the sub-term is an abstraction, then the $\delta\gamma$ rule will create duplicators to duplicate the sub-term (which has no Δ's in it so that it can be duplicated by Δ).

Example 2. We can now duplicate non-linear terms, such as $\lambda\, x\, .\, x\, x$

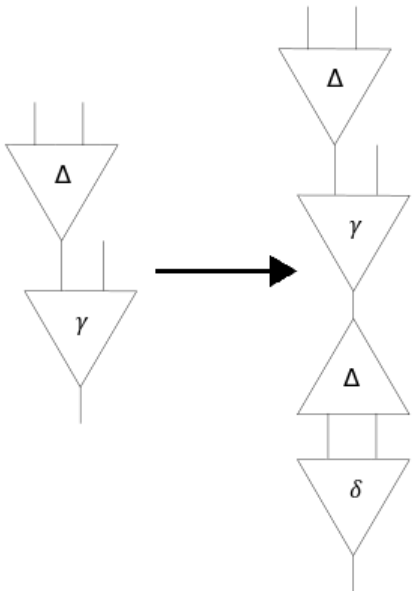


5.1 Duplicating Applications

Only one issue remains: what if a Δ reaches an application γ and no Δ ever reaches the principal port of that γ? We cannot simply take the approach from section 2.1 like Full Parallel Reduction does and move past these Δ's because the term, when normalized, might not actually be a well formed encoding of a term such as in the case of $(\lambda\, f\, .\, \lambda\, a\, .\, f\, (f\, (f\, a)))(\lambda\, g\, .\, \lambda\, b\, .\, g\, (g\, b))$



However, this can be solved by introducing a special reduction rule:



Where this rule may only be applied when the net which these cells are a part is a normalized net (because otherwise we could just be waiting for a Δ to reach the γ like in the duplication of a closed term). This can be implemented by keeping a global set of all Δ cells, and then once the net has been normalized, apply the rule to all of the Δ's attached to the right port of a γ, and then normalizing again (and repeat).

References

[1] Yves Lafont, *Interaction combinators*, Information and Computation **137** (1997), no. 1, 69 – 101.

[2] Ian Mackie and Jorge Sousa Pinto, *Compiling the lambda-calculus into interaction combinators*, 1998.