Δ-Nets: Interaction-Based System for Optimal Parallel λ-Reduction

Daniel Augusto Rizzi Salvadori

Abstract

I present a model of universal parallel computation called Δ-Nets, and a method to translate λ-terms into Δ-nets and back. Together, the model and the method constitute an algorithm for optimal parallel λ-reduction, solving the longstanding enigma with groundbreaking clarity. I show that the λ-calculus can be understood as a projection of Δ-Nets—one that severely restricts the structure of sharing, among other drawbacks. Unhindered by these restrictions, the Δ-Nets model opens the door to new parallel programming language implementations and computer architectures that are more efficient and performant than previously possible.

Interactive Demo and Source Code

`https://deltanets.org`

`https://github.com/danaugrs/deltanets`

1. Introduction

The λ-calculi are beautifully simple yet powerful models of universal computation. Consisting of *abstractions*, *variables*, and *applications*, λ-terms can express any computable function [Chu36, Tur36, Tur37, Chu41]. In addition to being central pillars in computation theory, the λ-calculi also constitute practical frameworks underpinning all functional programming languages. Four λ-calculi are pertinent to this paper—the three substructure λ-calculi [Jac93], and the full λ-calculus [Bar84]:

- **λL-calculus**: the *linear* λ-calculus, in which every bound variable occurs exactly once.
- **λA-calculus**: the *affine* λ-calculus, in which every bound variable occurs either once or not at all.
- **λI-calculus**[1]: the *relevant* λ-calculus, in which every bound variable occurs at least once.
- **λK-calculus**: the *full* λ-calculus in which bound variables can occur any number of times.

The λL-calculus can be regarded as a cornerstone which can be extended in one of three ways: with *erasure* (analogous to *weakening* in logic), resulting in the λA-calculus; with *sharing* (analogous to *contraction* in logic), resulting in the λI-calculus; or with *both erasure and sharing*, resulting in the full λK-calculus. Additionally, the λK-calculus can also be obtained by extending the λA-calculus with sharing, or the λI-calculus with erasure. These relationships are illustrated in Figure 1.

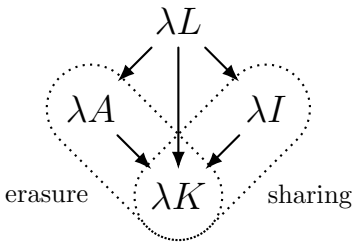

Figure 1: The relationships between the four λ-calculi in terms of erasure and sharing.

In the λ-calculi, a reduction strategy is *optimal* if and only if it reaches the normal form (if it exists) without perfoming any unnecessary reduction steps [Lev78, Lev80]. There are two types of unnecessary reductions:

1. Reduction of a later-discarded subexpression.
2. Reduction of a duplicated subexpression.

The first type can only occur with erasure, i.e., when some abstractions do not make use of their bound variables. Whenever such an abstraction is applied, the argument is discarded, and any reductions previously performed in the argument subexpression are rendered unnecessary. Naturally, this cannot happen in the λL-calculus nor in the λI-calculus. This first type of unnecessary reduction can be entirely avoided in the λA-calculus and in the λK-calculus by adhering to some reduction orders, including normal order reduction [Bar87].

The second type of unnecessary reduction can only occur with sharing, i.e., when some abstractions' bound variables occur multiple times. Whenever such an abstraction is applied, the argument is duplicated, and any reducible expressions in the argument subex-

[1] the original λ-calculus defined by Church [Chu36, Chu41].

pression are duplicated with it. Naturally, this cannot happen in the λL-calculus nor in the λA-calculus. Lévy has shown that there are λ-terms with sharing for which no reduction order is optimal [Lev78, Lev80]. The λ-calculus, as a sequential substitution machine, is therefore inadequate to express optimal reduction for all λ-terms. Does a more fundamental model of computation exist which is able to express optimal reduction for all λ-terms?

In order to avoid sharing-related unnecessary reductions in the λI- and λK-calculi, it is useful to represent λ-terms as graphs. Identical subexpressions can then be represented by the same shared subgraph, which only needs to be reduced once. The process of reduction is then expressed as a sequence of graph operations—a technique known as graph reduction [Wad71]. In the λ-calculi, applying a function destructively modifies its body. This presents a challenge in graph reduction when a function is shared, since it may be applied any number of times, each with a different argument. A simple solution is to duplicate the shared function's entire subgraph before applying it. This solution, however, leads to the second type of unnecessary reduction because it duplicates reducible expressions. It's possible to mitigate the number of duplicated reducible expressions, and thus of unnecessary reductions, by sharing, instead of duplicating, the function's *maximal free subgraphs*—the largest subgraphs in the function's body that don't make use of the function's bound variable [Wad71]. An ordering can also be imposed such that all reducible expressions inside a function are first reduced, and only then is the function duplicated (while sharing its maximal free subgraphs). However, a critical problem remains: this procedure never terminates in cyclic graphs, and cyclic graphs can arise from non-cyclic ones through regular reduction [Wad71]. In [Wad71] this is resolved by duplicating subgraphs whenever necessary to avoid cycles.

The first algorithms for λ-calculi reduction which don't perform any unnecessary β-reductions were proposed in [Lam89] and [Kat90]. The common core idea introduced was that of interior sharing of subgraphs. Interior sharing enables the incremental duplication of shared functions, which, in turn, fully prevents sharing-related unnecessary β-reductions. The challenge of interior sharing lies in the management, throughout reduction, of multiple simultaneous sharing contexts, each of which can fully or partially overlap any number of others. Moreover, sharing contexts can be recursive—a shared term can be referenced from inside itself any number of times. In [Lam89], interior sharing is accomplished through the use of explicit fan-in and fan-out nodes. Remarkably, the complex challenge then boils down to a simple question: when a particular fan-in meets a particular fan-out, should they annihilate one another or duplicate one another? In [Lam89], this is solved by associating a non-negative integer with each fan and introducing three delimiter node types to regulate fan numbers through graph-rewriting rules. When two fans meet, they are annihilated if and only if their numbers are equal, and they duplicate one another otherwise (Figure 2). The delimiters realize a notion of enclosure around fans, ultimately ensuring that only fans belonging to the same enclosure annihilate one another.

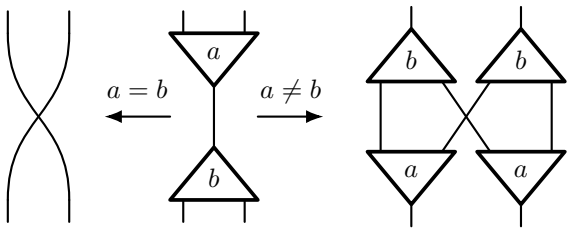

Figure 2: Interacting fans annihilate one another if they are equal, otherwise they duplicate one another.

Concurrently, a graphical meta-model of parallel computation called *interaction nets* was introduced in [Laf90]. It was later refined and expanded in [Laf97], once its significance was understood better. Models adhering to this framework are called *interaction systems*. An interaction system is specified by a set of agent types and a set of interaction rules—local graph rewriting rules that operate on pairs of connected agents. Agents are nodes that have one *principal* port and any number of *auxiliary* ports, including zero. When two agents are connected via their principal ports they form an *active pair*, and an interaction rule can then be applied to that pair. Since each agent can only be part of a single active pair at a time, interaction systems possess a one-step diamond property, which I denote as *perfect confluence*[2]. An extraordinary consequence of perfect confluence is that every normalizing interaction order produces the same result in the same number of interactions. Additionally, since interaction rules are local, they can be applied simultaneously without synchronization. These properties make interaction systems highly suitable for expressing optimal parallel algorithms.

An interaction system for λ-calculi reduction based on [Lam89], along with a translation method from and to λ-terms, was proposed in [GAL92a, GAL92b].

[2]Some authors denote this as *strong confluence*, but strong confluence has historically denoted a weaker property which is not one-step. I propose using the term *perfect confluence* for the one-step variant to avoid confusion.

A similar interaction system, combined with a different translation method, was presented in [AG98]. These interaction systems employ indexed agents called *brackets* and *croissants*, which increment and decrement, respectively, the index of higher-index agents they interact with. As explained in detail in [AG98], there are many ways to translate λ-terms into interaction nets using brackets and croissants. These agents effectively delimit sharing scopes, and, as such, are also referred to as *delimiters*. A comprehensive analysis and comparison of the computational efficiency of these algorithms, including the original [Lam89], was supplied in [LM99]. Unfortunately, all these algorithms have frustrating characteristics that profoundly undermine reduction performance. Critically, delimiters accumulate during reduction until delimiter interactions completely overwhelm fan interactions. Additionally, delimiters are often present in nets associated with λ-terms that have no sharing, serving no purpose there.

A more recent interaction system for λ-calculi reduction, called *Lambdascope*, was presented in [OL04]. It uses five agent types: an *abstractor*, an *applicator*, a *duplicator* (an indexed fan), an *eraser*, and an indexed delimiter. In Lambdascope, applying an abstraction effectively spawns two zero-indexed delimiters. This process, along with other interaction rules, preserves the scope of each abstraction after it has been applied. This *scope invariant*, in turn, ensures interacting duplicators annihilate and commute when appropriate. Delimiters move outward, expanding each scope as much as possible. Sibling scopes eventually coalesce, which helps reduce the number of total scopes and thus the number of total delimiters. However, siblingless scopes and their delimiters are preserved perpetually. As a result, Lambdascope also suffers from a devastating accumulation of delimiters. Additionally, since there's no mechanism to decrement indexes, they grow without bound as reduction progresses.

For example, when reducing the non-normalizing λ-term $(\lambda x.x\,x)(\lambda y.y\,y)$ with these algorithms, every iteration creates new delimiters. Each new delimiter takes up additional memory (space), and leads to additional unnecessary interactions (time). Additionally, the existing algorithms fail to establish a global reduction order, which is unfortunately critical to ensure that all nets associated with normalizing λ-terms normalize. Not only are the existing algorithms unsatisfactory from a theoretical perspective, their inefficiencies preclude them from being used at the core of programming language implementations.

In this paper, I present a model of universal parallel computation called Δ-Nets, and a method to translate λ-terms into Δ-nets and back. Together, the system and translation method constitute an algorithm for optimal parallel λ-reduction. As an interaction system, the core of the model is perfectly confluent: every normalizing reduction order produces the same result in the same number of steps. The core model is then necessarily extended with non-interaction *canonicalization* rules, which depart from the interaction paradigm. Along with a global reduction order, canonicalizations ensure Church–Rosser confluence and optimality in nonlinear systems. The Δ-Nets algorithm solves the longstanding enigma of optimal λ-calculi reduction with groundbreaking clarity. Instead of making use of delimiters, sharing is expressed through a single agent type which allows any number of auxiliary ports, called a *replicator*. Each instance of a replicator incorporates information that in previous models was spread across multiple agents, such as indexed fans and delimiters. This consolidation of information enables simplifications that were previously unfeasible, and leads to constant memory usage in the reduction of $(\lambda x.x\,x)(\lambda y.y\,y)$, for example. Finally, I show that the λ-calculus can be understood as a projection of Δ-Nets. The additional degrees of freedom in Δ-Nets allow it to realize optimal reduction in the manner envisioned by Lévy, i.e., no reduction operation is applied which is rendered unneccessary later, and no reduction operation which is necessary is applied more than once.

2. Core Interaction System

At its core, a Δ-net is an interaction net with three agent types: a *fan*, an *eraser*, and a *replicator*.

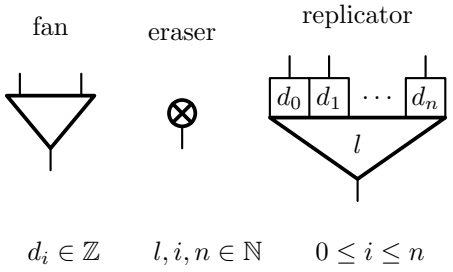

Figure 3: The three Δ-Nets agent types.

Fans have two auxiliary ports, and erasers have none. Each replicator can have a different natural number of auxiliary ports, and each of those ports has an associated integer called a *level delta*. Additionally, each replicator has an associated non-negative integer called a *level*. As a point of reference, an agent's auxiliary ports are ordered in a clockwise orientation, i.e., from left to right when the principal port points down.

Fans are sufficient to express optimal parallel reduction in the λL-calculus. Erasers are needed to express optimal parallel reduction involving erasure. Replicators are needed to express optimal parallel reduction involving sharing. As such, the Δ-Nets core interaction system decomposes perfectly into three overlapping subsystems, each analogous to a substructure λ-calculus. The full Δ-Nets system may also be referred to as ΔK-Nets. The four systems are:

- ΔL**-Nets**: the *linear* Δ-Nets subsystem, which only uses fans, and expresses optimal parallel reduction in the λL-calculus.
- ΔA**-Nets**: the *affine* Δ-Nets subsystem, which uses fans and erasers, and expresses optimal parallel reduction in the λA-calculus.
- ΔI**-Nets**: the *relevant* Δ-Nets subsystem, which uses fans and replicators, and expresses optimal parallel reduction in the λI-calculus.
- ΔK**-Nets**: the *full* Δ-Nets system, which uses fans, erasers and replicators, and expresses optimal parallel reduction in the λK-calculus.

When equal agents interact, they *annihilate* one another. In general, two replicators are equal if and only if they have the same level, number of auxiliary ports, and level deltas. However, when a Δ-net is constructed from a λ-term via the translation method presented in the next section, interacting replicators that have the same level are guaranteed to be equal. As such, only replicator levels need to be compared for equality in practice. Equal-agent interactions are called *annihilations*.

When distinct agents interact, each agent travels through and past the other and is potentially copied or erased in the process, depending on how many auxiliary ports the other agent has. Since an eraser has no auxiliary ports, it *erases* every agent it interacts with. As such, distinct-agent interactions involving erasers

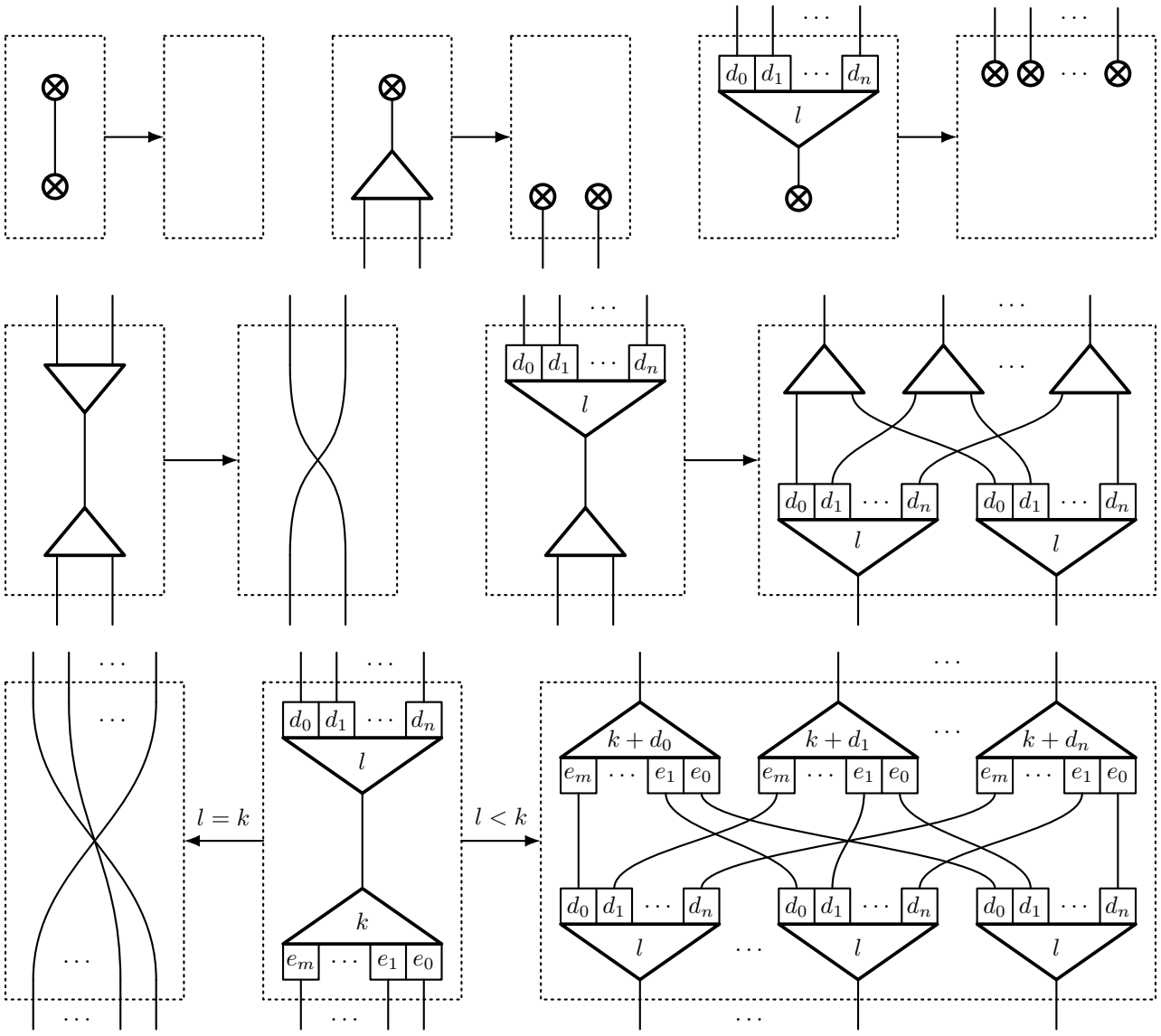

Figure 4: The core Δ-Nets interaction rules.

are called *erasures*. The remainder of distinct-agent interactions are called *commutations*. When a replicator interacts with a fan, the replicator travels through and out of the fan's two auxiliary ports, resulting in two exact copies of the replicator. Simultaneously, the fan travels through and out of all of the replicator's auxiliary ports, resulting in a fan for each replicator auxiliary port.

When two distinct replicators interact, the lower-level one replicates the higher-level one once for each of the lower-level one's auxiliary ports, while, simultaneously, the higher-level one duplicates the lower-level one once for each of the higher-level one's auxiliary ports. Note that whereas a *duplication* produces exact copies, a *replication* produces copies that may or may not be exact. Replicators are so named because the copies they produce of other replicators are, by design, not necessarily exact. Each resulting replica of the higher-level replicator may have a different level, determined by the level delta associated to the auxiliary port of the lower-level replicator that it travels out of. The level of each resulting replica is the sum of the level of the original higher-level replicator and the appropriate level delta of the lower-level replicator. The Δ-Nets interaction rules are illustrated in Figure 4.

3. From λ-terms to Δ-nets

Definition. *Let $\Sigma = \{L, A, I, K\}$. For all $S \in \Sigma$, let Λ_S be the set of all λS-terms and Δ_S be the set of all ΔS-nets. For all $S \in \Sigma$, there exists a bijection $\phi_S : \Lambda_S \to \Delta_S^c$ which maps every λS-term to a* canonical *ΔS-net.*

$$\Delta_S^c = \{\phi_S(\lambda_S) \mid \forall \lambda_S \in \Lambda_S\} \qquad \text{(canonical } \Delta S\text{-nets)}$$

$$\Delta_S^p = \{\delta_S^p \mid \forall \delta_S^c \in \Delta_S^c,\ \delta_S^c \xrightarrow{\Delta^*} \delta_S^p\} \quad \text{(proper } \Delta S\text{-nets)}$$

$$\Delta_S^c \subseteq \Delta_S^p \subset \Delta_S$$

The bijections ϕ_S, $\forall S \in \Sigma$, are defined inductively, with the rules for the general ϕ_K case illustrated in Figure 5. In the description that follows, the minor differences in ϕ_L, ϕ_A, and ϕ_I are noted where appropriate. Each outer dashed rectangle in Figure 5 contains a Δ-net fragment that represents the λ-term specified above it. Each inner dashed rectangle is a slot for the Δ-net that represents the inner term. Each thick vertical dashed line represents a non-negative integer number of parallel wires (including zero). Every dashed rectangle, outer or inner, has the same interface—a single wire entering at the top and a non-negative integer number of wires leaving at the bottom. Each λ-term has a subscript associated with it, which represents the *level* of that term. The level of the outermost term is set to zero, which inductively sets all other levels. The level of an application's argument is one greater than that of the application itself, and the level of a replicator is one greater than that of its associated abstraction. These levels are ultimately used to determine the level and level deltas of replicators.

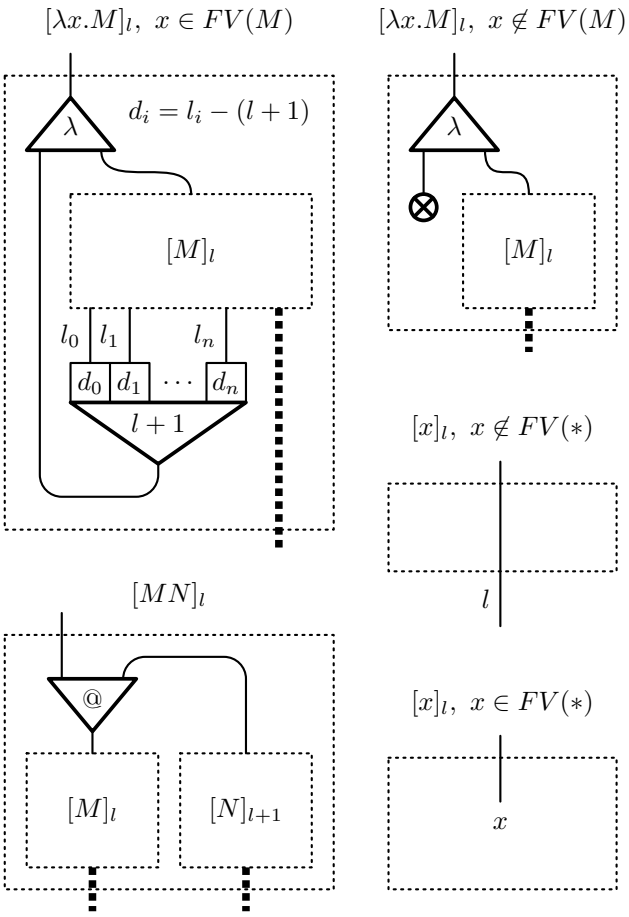

Figure 5: The inductive definition of ϕ_K, a bijection which translates λK-terms into canonical ΔK-nets.

There are two Δ-net fragments for variables, one for variables that are free in the outermost λ-term ($x \in FV(*)$), and one for variables that are not ($x \notin FV(*)$), i.e., that are bound by some abstraction. The free-variable fragment contains a single-port (non-agent) node, which is represented by the name of the associated free variable in the λ-term. The bound-variable fragment is just a vertical wire. An instance of the bound-variable fragment which represents the ith occurrance of a bound variable in the associated λ-term has its bottom wire endpoint connected to the ith auxiliary port of the replicator that shares that variable. The bottom wire endpoint of each bound-variable fragment has a level associated with it, which is equal to the level of the variable. These wire endpoint levels are referenced in the fragment that fea-

tures a replicator, in the definition of the level deltas d_i. The level delta associated to an auxiliary port of a replicator is equal to the level of the wire connected to that auxiliary port minus the level of the replicator.

There are also two Δ-net fragments for abstractions, one for those which use their bound variables and one for those which don't. In both, the abstraction itself is represented by a fan pointing up, labelled with a λ[3]. The principal port of the fan is the parent port of the abstraction. The first auxiliary port of the fan is a child port, connected to the Δ-net that represents the body of the abstraction. The second auxiliary port represents the variable of the abstraction, and since it is connected to the parent of the variable, it is another parent port. If an abstraction doesn't use its bound variable, the fan's second auxiliary port is connected to an eraser. The fragment with this eraser is only needed when translating expressions from λ-calculi with erasure, and, as such, it is exclusively part of ϕ_A and ϕ_K. When translating expressions from λ-calculi with sharing, if an abstraction uses its bound variable, then the fan's second auxiliary port is generally connected to the principal port of a replicator, which shares the abstraction's variable among the bound-variable fragments that represent its various occurances in the λ-term. However, in a canonical Δ-net, a replicator with a single auxiliary port and a level delta of zero, regardless of the replicator's level, is equivalent to a wire. Therefore such replicators are never created in the first place, and the appropriate bound-variable fragment's bottom wire is connected directly to the abstraction fan's second auxiliary port instead. In ϕ_L and ϕ_A, which translate λ-terms without sharing, the fragment with the replicator is modified such that the replicator is substituted by a single wire in the same way.

Finally, there is a fragment for applications, in which the application itself is represented by a fan pointing down, labelled with an @[3]. The first auxiliary port of the fan is the parent port of the application. The principal port of the fan is a child port, connected to the Δ-net that represents the function λ-term. The second auxiliary port is another child port, connected to the Δ-net that represents the argument λ-term. In addition to the fragments shown in Figure 5, every canonical Δ-net has a single *root* (non-agent) node, represented by a small circle, connected to the top of the outermost Δ-net fragment. Together, the root node and the free variable nodes constitute the interface of a canonical Δ-net.

Note that both applications and abstractions are represented by fans, and β-reduction is expressed through fan annihilation. Whether a specific fan in a Δ-net represents an abstraction or an application can always be determined by inspecting the net (without fan labels) and tracing paths from the root node. While abstraction fans have two parent ports and one child port, application fans have one parent port and two child ports. Every port of any node (including non-agent nodes) in a proper Δ-net is either a child port or a parent port, and every wire connects a child port with a parent port. Although this duality has no direct bearing on the interaction process, it is clearly present, and it is instrumental in some discussions.

All replicators in a canonical Δ-net are unpaired fan-ins: each auxiliary port is a parent port and the principal port is a child port. However, during reduction, fan-out replicators may be produced. In a fan-out replicator, the principal port is a parent port and each auxiliary port is a child port. Every commutation between a fan and a replicator (either a fan-in or a fan-out) always produces a fan-in and a fan-out. While every fan-out is paired with at least one upstream fan-in, the converse is not true: fan-ins may or may not be paired. Locally determining this pairing efficiently is the purpose of the level delta system.

In some ways, a fan is just a replicator with two auxiliary ports, zero level, and zero deltas. If replicators were allowed to have zero auxiliary ports, then an eraser could also just be a replicator with no auxiliary ports and zero level (or no level). It's tempting to attempt to consolidate the three agents into one—the Δ-agent, if you will. However, fans and replicators have a critical distinction in the types of their auxiliary ports: a replicator's auxiliary ports are either all parent ports or all child ports, while a fan has one of each. This affects how replicator pairedness is tracked: after fan replication the resulting replicators become paired whereas in replicator replication they keep their original status.

In standard Δ-Nets, replicators have *absolute levels*, i.e., the initial level of each replicator is exactly determined by how many times a path from the root to the replicator's principal port traverses fans out of their second auxiliary ports. There exists a dual formulation of the Δ-Nets system to this where replicators have *relative* levels—all replicators start with level zero, and a replicator's level gets incremented when it traverses out of the second auxiliary port of a fan.

[3]Fan labels are just a visual aid and do not affect how fans interact with other agents.

4. From Δ-nets back to λ-terms

The only interaction rule in ΔL-Nets is fan annihilation, which expresses β-reduction. Therefore if n β-reductions normalize a λL-term t, then n interactions normalize the λL-net $\phi_L(t)$. As a result, in the ΔL-Nets system, all proper nets are canonical, and fan annihilations can be applied in any order, with perfect confluence. Compared to ΔL-Nets, the other Δ-Nets systems have additional interaction rules, which don't have λ-calculus analogues.

In Δ-Nets systems with erasure, applying an abstraction which doesn't use its bound variable results in an eraser becoming connected to a parent port. Such an eraser could erase abstraction fans and fan-out replicators, but if it reaches the parent port of an application fan, for example, which is an auxiliary port, the erasure process would cease. As a result, absent additional rules, irreducible subnets would be produced which don't have a λ-calculus analogue. As an extreme example, applying an abstraction which doesn't use its bound variable to an argument which only uses globally-free variables produces a subnet which is disjointed from the root. In order to eliminate all such subnets a final *canonicalization* reduction step is introduced in Δ-Nets systems with erasure: all parent-child wires starting from the root are traversed and nodes are marked. All non-marked nodes are then erased, and wires that were connected to these nodes are instead connected to erasers. This final canonicalization erasure step dispenses with the need to apply erasure and eraser annihilation rules. In fact, this step can be applied at any point during reduction in order to reduce the net size, effectively trading computation (time) for memory (space). In order to keep memory usage to a minimum, this step should be applied after every application of an abstraction which doesn't use its bound variable.

In order to ensure that no reduction operations are applied in a subnet that is later going to be erased, a sequential leftmost-outermost reduction order needs to be followed. Therefore, in the ΔA-Nets system, fan annihilations are applied in leftmost-outermost order, with the final erasure canonicalization step ensuring perfect confluence, and producing a normal canonical ΔA-net.

During reduction of ΔI- and ΔK-Nets, replicators can combine into *replicator trees*. A replicator tree is a subnet containing only replicator agents such that each one's principal port is connected to an auxiliary port of another, except for the tree's *root* replicator. In general, replicators that belong to the same tree cannot be merged during reduction because they may be paired with other replicators elsewhere. However, if consecutive replicators in a replicator tree are known to be unpaired, they can be safely merged together. *Unpaired replicator merging* is a canonicalization rule present in both ΔI- and ΔK-Nets. Additionally, in the ΔK-Nets system, it is possible to eliminate unpaired replicators' auxiliary ports which are connected to erasers—a canonicalization rule called *unpaired replicator decay*. As part of unpaired replicator decay, if an unpaired replicator is left with a single auxiliary port with a level delta of zero, it is replaced by a wire, as it is equivalent to one. This canonicalization rule is applied to all unpaired replicators as part of the erasure canonicalization step. It can also be lazily applied to the involved unpaired replicator, immediately before the fan replication, replicator replication, and aux fan replication rules.

Replicators start as *unpaired*, and this status is propagated across interactions. When an unpaired replicator interacts with a fan, the status of both resulting replicators changes to *unknown*. If an unpaired replicator (A) is connected to a consecutive replicator (B) of unknown status via an auxiliary port, and a certain local constraint is met, then the consecutive replicator can be determined to be unpaired, and the two can then be merged. The constraint is met when the second replicator's level is greater than or equal to the first replicator's level, but no greater than the first replicator's level plus the level delta of the auxiliary port that connects them: $0 \leq l_B - l_A \leq d$. Under this constraint, no replicator is able to interact with the second replicator before the first replicator is annihilated. Since the first replicator is unpaired, it can never be annihilated, and the second one must be unpaired as well.

It is possible to limit replicators to have at most two auxiliary ports, or even exactly two auxiliary ports. This imposes the smallest possible upper bound on interaction rule complexity and agent size, making the total number of interactions an effective measure of time complexity, and the total number of agents an effective measure of space complexity. A tree of such replicators can stand in for any replicator with any number of auxiliary ports.

Replicator merging is a canonicalization rule and not an interaction rule because it involves two agents that are connected via ports that aren't both principal. Merging replicators as early as possible reduces the total number of reductions and the total number of agents, improving space and time efficiency. The reduction order which guarantees that replicator merges happen as early as possible, minimizing the total number of reductions, is a sequential leftmost-outermost

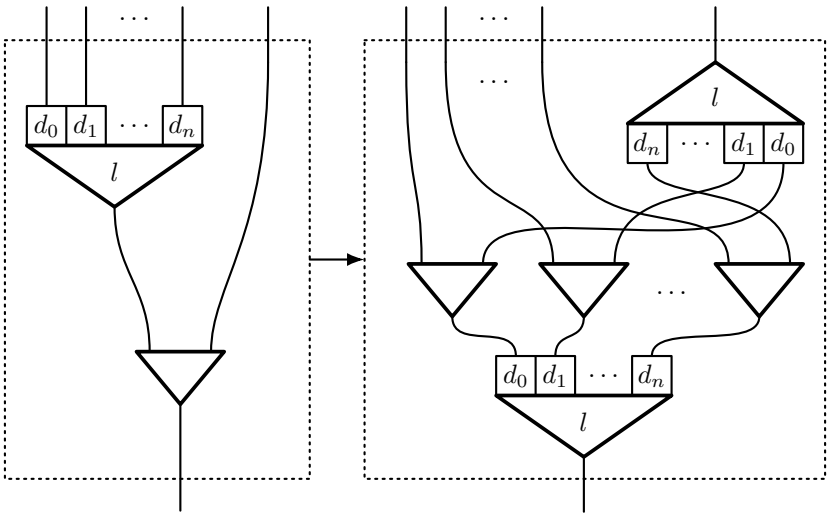

Figure 6: The aux fan replication canonicalization rule in ΔI- and ΔK-Nets.

order—the optimal reduction order for ΔA-, ΔI- and ΔK-Nets. In fact, any reducible pair that eventually reaches the leftmost-outermost position unchanged can be reduced at any time. For example, all potential annihilations in the spine that don't involve unpaired replicators can be applied in any order, because they will eventually reach the top of the spine unchanged. On the other hand, a potential commutation that involves an unpaired replicator may not reach the top of the spine unchanged, as the involved replicator may be merged with another beforehand, rendering its early application suboptimal.

To see how leftmost-outermost reduction is optimal in ΔI- and ΔK-Nets, observe that no commutation involving an unpaired replicator can be applied before that replicator is merged—if merging it is at all possible. Take a leftmost-outermost interaction between two replicators. The fan-out replicator is necessarily paired. If the fan-in replicator is unpaired and can eventually be merged, then the replicator merging—or any intermediate reductions leading to it—would occur higher in the spine and would be applied first. This reasoning holds even in the presence of loops. In ΔK-Nets, the leftmost-outermost order is critical not only to achieve optimality but also to ensure that all nets associated with normalizing λ-terms normalize. As for the remaining rules, the Δ-Nets systems inherit their optimality guarantee from the interaction nets paradigm.

Additionally, in ΔI- and ΔK-Nets, the reduction process needs to be split in two phases. In the first phase the core interaction rules and unpaired replicator merging are applied in leftmost-outermost order until no further reduction can be applied. The reduction process then switches to the second phase, in which the *aux fan replication* rule (illustrated in Figure 6) replaces the core fan replication rule. This second phase can be alternatively understood as modifying all fans such that the first auxiliary port becomes the principal port. This process transforms the sharing structures so that all fan-out replicators are eliminated, all appropriate subnets are replicated, and all fan-in replicators accumulate at the variable port of abstraction fans. The result is a canonical ΔI-net or, after the final erasure canonicalization step, a canonical ΔK-net. Since all normal Δ-nets are canonical, the Δ-Nets systems are all Church–Rosser confluent.

Definition. *For all $S \in \Sigma$ and for all $\lambda_S \in \Lambda_S$, if λ_S is normalizing, λ_S β-reduces to $\phi_S^{-1}(\Omega_S(\phi_S(\lambda_S)))$ where $\Omega_S : \Delta_S^p \to \Delta_S^c$ reduces a proper ΔS-net through interaction and canonicalization rules as defined in this section until it is normal and canonical.*

Theorem. *Since $\phi_S^{-1} \circ \Omega_S$ is idempotent $\forall S \in \Sigma$, the set of all λS-terms, Λ_S, is a projection of proper ΔS-nets:*

$$\Lambda_S = \{\phi_S^{-1}(\Omega_S(\delta_S^p)) \mid \forall \delta_S^p \in \Delta_S^p\},\ \forall S \in \Sigma$$

Moreover, since Λ_S is closed under β-reduction and Δ_S is closed under Δ_S-interactions and canonicalizations, the λS-calculus can be interpreted as a projection of ΔS-Nets, $\forall S \in \Sigma$.

5. Conclusion

The existence of nonlinear proper Δ-nets which are not canonical reflects the additional degrees of freedom that interior sharing introduces, which are not present in the λ-calculus. A given λI-term can potentially be represented by many different proper ΔI-nets, which differ only with respect to their sharing structure. As an example, take a proper λI-net n, with $\phi_I^{-1}(\Omega_I(n)) = MM$, for some λI-term M. The

λI-net n may have two distinct but equal subnets that each represent M, or it may have a single such subnet which is shared among the two occurrances in the application. In fact, the sharing structure of n could be arbitrarily complex. These additional degrees of freedom allow Δ-Nets to realize optimal reduction in the manner envisioned by Lévy, i.e., no reduction operation is applied which is rendered unneccessary later, and no reduction operation which is necessary is applied more than once.

References

[Chu36] A. Church, *An Unsolvable Problem of Elementary Number Theory*, 1936

[Tur36] A. M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, 1936

[Tur37] A. M. Turing, *Computability and λ-definability*, 1937

[Chu41] A. Church, *The Calculi of Lambda-Conversion*, 1941

[Wad71] C. P. Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, PhD Thesis, Oxford, 1971

[Lev78] J.-J. Lévy, *Réductions Correctes et Optimales dans le Lambda-Calcul*, Doctorat d'État, Paris 7, 1978

[Lev80] J.-J. Lévy, *Optimal Reductions in The Lambda-Calculus*, 1980

[Bar84] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, 1984

[Bar87] H. P. Barendregt, J. R. Kennaway, J. W. Klop, M. R. Sleep, *Needed Reduction and Spine Strategies for the Lambda Calculus*, 1987

[Lam89] J. Lamping, *An Algorithm for Optimal Lambda Calculus Reduction*, 1989

[Kat90] V. K. Kathail, *Optimal Interpreters for Lambda-calculus Based Functional Languages*, 1990

[Laf90] Y. Lafont, *Interaction Nets*, 1990

[GAL92a] G. Gonthier, M. Abadi, J.-J. Lévy, *The Geometry of Optimal Lambda-Reduction*, 1992

[GAL92b] G. Gonthier, M. Abadi, J.-J. Lévy, *Linear Logic Without Boxes*, 1992

[Jac93] B. Jacobs, *Semantics of lambda-I and of other substructure lambda calculi*, 1993

[Laf97] Y. Lafont, *Interaction Combinators*, 1997

[AG98] A. Asperti, S. Guerrini, *The Optimal Implementation of Functional Programming Languages*, 1998

[LM99] J. L. Lawall, H. G. Mairson, *Optimality and inefficiency: what isn't a cost model of the lambda calculus?*, 1999

[OL04] V. van Oostrom, K.-J. van de Looij, *Lambdascope*, 2004