Dance of the Solos

On the graphical representation of Solo Diagrams

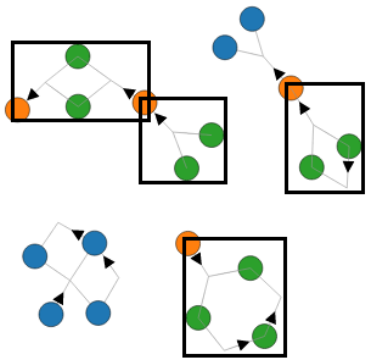
`https://github.com/AdamLassiter/solo-calc`

Adam Lassiter

May 2018

Abstract

The project seeks to provide an implementation of both the Solo Calculus and also of Solo Diagrams, an intuitively graphical representation of the calculus. These diagrams can then aid in understanding of reductions of process calculi, while maintaining a clean and simple language with strong provable properties.



Preface

Before you lies the culmination of a year's work on the weird and wonderful Solo Calculus. Written in support of MComp Computer Science with Mathematics degree at University of Bath, it has been a joy to see the project come to life throughout the last eight months. I hope the realisations and results found within offer the same new perspective on the workings of an implemented process calculus as they have given me.

I would like to express my gratitude to Prof. Guy McCusker for his guidance and dedicated supervision of this project throughout. Further thanks are extended to my mother, whose constant support and advice on the politics and practices of academic writing have proven invaluable.

Happy reading,

Adam Lassiter

Contents

1 Introduction

1.1 Motivation, Landscape and Current Problem

As computer processor clock speeds have begun to stagnate in performance increase per year, manufacturers have begun to shift focus to multiple cores in search of greater performance. Supercomputers for decades have operated around the idea of clusters and massive parallelism. For this reason, there is now more than ever a need to study the effects of parallelism in computation and to be able to effectively model concurrent communicating systems as they start to become commonplace in all computational applications.

1.2 Scope and Relevance of Research

The following aims to provide a visualisation of the parallel computation of multiple processes within the context of a concurrency calculus. This will be divided into several modular parts that apart demonstrate some challenges on successful implementation, but together form what should prove to be a useful proof-of-concept tool to aid in understanding of calculi of communicating processes.

In particular, the project tackles the details of implementation of a computational calculus of communicating mobile processes. This implementation is then further extended to provide an interactive visualisation of computations to aid in the understanding of how such calculi operate. As discussed later in 2.3, the project aims to provide an improvement over the π-calculus described by Milner (1999), while still remaining functionally equivalent.

1.3 Project Aims

The project seeks to provide an implementation of both the Solo calculus (see Laneve and Victor (1999)), for which there exists an encoding of the π-calculus within itself, and also of Solo diagrams (see Laneve et al. (2001)), an intuitively graphical representation of the calculus. These diagrams can then aid in understanding of reductions of process calculi, while maintaining a clean and simple language with strong, provable properties. Furthermore, the design of the Solo calculus is such that it is fully asynchronous. That is, unlike in the π-calculus where processes block while waiting for inputs/outputs, it is shown through the construction of the Solo calculus that no such system is required however there still exists a way of building such a system to give the effects of the π-calculus should it be desired. For these reasons, the Solo calculus is found to be an interesting alternative to the more common π-calculus.

1.4 Overview of Dissertation Structure

The first section details an in-depth review of the surrounding literature and current state of the art. This includes an examination of various computational calculi, both concurrent and not, and an evaluation of their effectiveness for their given use-cases. Included also are some short examples to give a feel of how each calculus is used.

Afterwards follows a short investigation on technologies planned to be used in this project and justifications as to why each was chosen.

Following that is then a breakdown of the development process, description of some simple algorithms involved and the pitfalls which may not be immediately obvious. This section attempts to show both the initial and final forms of the project and what difficulties caused this evolution.

Finally, a short conclusion of effectiveness of the design choices made and how a similar project could be conducted differently.

2 Literature

2.1 λ-calculus

Developed by Alonzo Church in the 1930s, the λ-calculus was the first such computational calculus and describes a mathematical representation of a computable function. When it was first designed, it was not expected to be as relevant to the newly-emerging field of theoretical Computer Science but instead Discrete Mathematics and Category Theory, the λ-calculus in fact forms a universal model of computation and can contain an encoding of any single-taped Turing machine. It has since become a popular formal system within which to study properties of computation.

Definition 2.1.1. Syntax
The λ-calculus, as defined by Church but here explained by Machado (2013)* consists of an expression M built from the following terms:

$$\begin{array}{rcll} M & := & a & \text{(variable)} \\ & & \lambda x.M & \text{(abstraction)} \\ & & MN & \text{(application)} \end{array}$$

From this, any computable function can be constructed and computation is achieved through a series of operations on the expression.

Definition 2.1.2. α-substitution
Unbound variables within an expression may be substituted for any given value. This is formally expressed as:

$$x[y := P] := \begin{cases} P & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x.M)[y := P] := \begin{cases} \lambda x.M & \text{if } x = y \\ \lambda x.(M[y := P]) & \text{if } x \neq y \text{ and } x \notin FV(P) \ ^{\dagger} \end{cases}$$

This operation may be thought of as variable renaming as long as both old and new names are free in the expression in which they were substituted.

*While Church's original paper is still available, the source cited is found to be more relevant due to research in the subject area since the original paper's publication in the 1930s.

†Here, $FV(P)$ is the set of all variables x such that x is free (unbound) in P.

Corollary. α-equivalence
The above definition of α-substitution may be extended to give an equivalence relation on expressions, α-equivalence, defined as:

$$y \notin FV(M) \implies \lambda x.M \equiv_\alpha \lambda y\,(M[x := y])$$
$$M \equiv_\alpha M' \implies \begin{cases} M\,P \equiv_\alpha M'\,P \\ P\,M \equiv_\alpha P\,M' \\ \lambda x.M \equiv_\alpha \lambda x.M' \end{cases}$$

Definition 2.1.3. β-reduction
An expression may be simplified by applying one term to another through substitution of a term for a bound variable. This is formally expressed as:

$$(\lambda x.P)\,Q \to_\beta P[x := Q]$$
$$M \to_\beta M' \implies \begin{cases} P\,M \to_\beta PM' \\ M\,P \to_\beta M'P \\ \lambda x.M \to_\beta \lambda x.M' \end{cases}$$

Often β-reduction requires several steps at once and as such these multiple β-reduction steps are abbreviated to $\to_\beta^*$.

Corollary. β-equivalence
The above definition of β-reduction may be extended to give an equivalence relation on expressions, *beta*-equivalence, defined as:

$$M \to_\beta^* P \text{ and } N \to_\beta^* P \implies M \equiv_\beta N$$

Example. The above corollaries can be seen to have desirable properties when examining whether two expressions describe equivalent computation.

$$\lambda a.x\,a[x := y] \equiv \lambda a.y\,a$$

$$\lambda x.x\,y \equiv_\alpha \lambda z.z\,y$$
$$\lambda x.x\,y \not\equiv_\alpha \lambda y.y\,y$$

$$(\lambda a.x\,a)\,y \to_\beta x\,a[a := y] \equiv x\,y$$

As computational calculus shares many parallels with modern functional programming, the following are encodings of some common functional concepts within the λ-calculus.

Definition 2.1.4. List
Within the λ-calculus, lists may be encoded through the use of an arbitrary *cons* function that takes a head element and a tail list and of a *null* function that signifies the end of a list. The list is then constructed as a singly-linked list might be constructed in other languages:

$$[x_1, \ldots, x_n] := \lambda c.\lambda n.(c\,x_1\,(\ldots(c\,x_n\,n)\ldots))$$

Definition 2.1.5. Map
The *map* function takes two arguments — a function F that itself takes one argument and a list of suitable arguments $[x_1, \ldots, x_n]$ to this function. The output is then a list of the output of F when applied to each $x_1 \ldots x_n$.

$$map := \lambda f.\lambda l.(\lambda c.(l\,(\lambda x.c\,(f\,x))))$$

Example. As follows is an example of the reductions on the *map* function for a list of length $n := 3$:

$$\begin{aligned}
map\,F\,[x_1, x_2, x_3] &\equiv_\alpha \lambda f.\lambda l.(\lambda c.(l\,(\lambda x.c\,(f\,x))))\,F\,\lambda c.\lambda n.(c\,x_1\,(c\,x_2\,(c\,x_3\,n))) \\
&\to^*_\beta \lambda c.(\lambda c.\lambda n.(c\,x_1\,(c\,x_2\,(c\,x_3\,n))))\,(\lambda x.c\,(F\,x))) \\
&\to^*_\beta \lambda c.(\lambda n.((\lambda x.c\,(F\,x))\,x_1\,((\lambda x.c\,(F\,x))\,x_2\,((\lambda x.c\,(F\,x))\,x_3\,n)))) \\
&\to^*_\beta \lambda c.(\lambda n.(c\,(F\,x_1)\,(c\,(F\,x_2)\,(c\,(F\,x_3)\,n)))) \\
&\equiv_\alpha [F\,x_1, F\,x_2, F\,x_3]
\end{aligned}$$

Remarks. While the λ-calculus has been successful and been studied by various areas of academia outside of Computer Science, it is limited in modern-day application by its fundamentally 'single-process' model and struggles to describe multiple systems working and communicating together. While certain additional properties — specifically the Y-combinator and simply-typed λ-calculus — are not mentioned here, the calculus is defined from a few simple rules. This simplicity allows implementations of λ-calculus interpreters to be relatively painless.* *In my opinion, the simplicity and expressiveness of the λ-calculus should be the standard to which other computational calculi are held.*

The *map* example is particularly relevant to study within concurrent calculi as multiple large-scale systems follow a MapReduce programming model, as described by Dean and Ghemawat (2008), which utilises massive parallelism of large detacenters. The model requires a *map* function that applies a function to a key and set of values, similar to that described above, and a *reduce* function that collects all values with matching keys. This has found to be useful for modelling many real-world tasks for performance reasons, but concurrent calculi may provide a simple case for study and understanding for any of these tasks which scales as necessary.

2.2 Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS), as described by Milner (1980), was one of the earlier† process calculi. It was designed in the same vein as Church's λ-calculus, but with a focus on modelling concurrent systems. Among the many differences, most notable are the ability for concurrency and synchronisation through waiting on input/output through names. For reasons discussed in 2.3, it did not become as mainstream as the λ-calculus but

*There exists an example of such an interpreter, available online at the time of writing, at `http://www.cburch.com/lambda/`

†The reader is referred to 'Communicating Sequential Processes' by Hoare (1978), which may be thought of as the first such concurrent process calculus. It is excluded here as it bears more similarities with a traditional programming language and is therefore less relevant.

did serve as an important basis for study in the subject distinct from Church's single-process model.

Definition 2.2.1. Syntax
Within CCS, a process P is defined as:

P	:=	nil or 0	(inaction process)
		x	(variable)
		τ	(silent action)
		$ax_1 \dots x_n$	(action on $x_1 \dots x_n$) *
		$P \mid Q$	(composition)
		$P + Q$	(choice / summation)
		$P \backslash a$	(restriction)
		$P[b/a]$	(relabelling, a := b) †
		$x(x_1 \dots x_n)$	(identifier)
		if x then P else Q	(conditional)

When comparing to the λ-calculus from Definition 2.1.1, certain parallels can be seen. Notable additions are the action operator and the composition operators (parallel composition and choice).

Due to the increased amount of syntax, CCS has many more rules and semantics for computation than the λ-calculus. Here $\xrightarrow{a}$ describes reduction through taking an arbitrary action a.

Definition 2.2.2. Action Semantics
Given two processes waiting on input/output, eventually an input/output action will happen. This gives the reductions as follows:

$$ax_1 \dots x_n.P \xrightarrow{av_1 \dots v_n} P[v_1/x_1 \dots v_n/x_n]$$
$$\bar{a}v_1 \dots v_n.P \xrightarrow{\bar{a}v_1 \dots v_n} P$$
$$\tau.P \xrightarrow{\tau} P$$

Definition 2.2.3. Composition Semantics
For such an action a taking place, the following reductions can be made:

$$P \xrightarrow{a} P' \implies \begin{cases} P + Q \xrightarrow{a} P' \\ P \mid Q \xrightarrow{a} P' \end{cases}$$

$$\left.\begin{array}{l} P \xrightarrow{a} P' \\ Q \xrightarrow{\bar{a}} Q' \end{array}\right\} \implies P \mid Q \xrightarrow{\tau} P' \mid Q'$$

The choice here of whether to follow the left or right side is non-deterministic. This leads to what is described as the Tea/Coffee Problem.

Example. Tea/Coffee Problem
Suppose there is a machine that, when given a coin, will dispense either tea

These actions come in pairs a and $\bar{a}$ representing input and output respectively.

*There may be multiple relabellings at once, so this is often written $p[S]$ where the function S has $dom(S) = \{a\}\, ran(S) = \{b\}$

or coffee. A user comes to insert a coin to get some tea. This system could be described as:

$$coin\,.\,\overline{tea}\,.\,0 \;+\; coin\,.\,\overline{coffee}\,.\,0 \quad | \quad \overline{coin}\,.\,tea\,.\,0$$

But this would be incorrect. After the $\xrightarrow{coin}$ action, a choice would need to be made as to whether prepare to output tea or to output coffee. Only in the case that it is decided to output tea does the system halt successfully. Otherwise, it is left in the state:

$$\overline{coffee}\,.\,0 \quad | \quad tea\,.\,0$$

The problem would instead be successfully encoded as:

$$coin\,.\,(\overline{tea}\,.\,0 \;+\; \overline{coffee}\,.\,0) \quad | \quad \overline{coin}\,.\,tea\,.\,0$$

It is important here to note that the *trace* of both programs (the set of inputs that produce accepted outputs) is identical, but the two are not *bisimilar* (they are equivalent to the actions that can be taken at any step). These concepts will be examined further later.

Remark. A Note on Traces and Bisimulation
The idea of both *trace* and *bisimulation* are attempts to define a system for analysing *behavioural equivalence*. That is, given two programs that are written very different but behave similarly, is their behaviour exactly equivalent.

The trace comes from automata theory and is exactly equivalent to *language equivalence* — given a set of inputs, do both programs accept and reject (or produce the same output) for all the same inputs. Trace equivalence is considered the weakest equivalence for two systems.

Bisimulation however holds roots from a more mathematical standpoint — does there exist a bijection between behaviours of each system *at any given step*. More specifically, two processes P, Q are bisimilar (written $P\,\mathcal{R}\,Q$) if:

$$\left.\begin{array}{l} P\,\mathcal{R}\,Q \\ P \xrightarrow{\alpha} P' \end{array}\right\} \implies \exists Q' : \begin{cases} Q \xrightarrow{\alpha} Q' \\ P'\,\mathcal{R}\,Q' \end{cases}$$

This is a much stronger property and is one of the strongest that can be shown except for α-equivalence (identical up to name-substitution).

From the Tea/Coffee Problem, it can be seen that the two systems are trace-equivalent. However, they are not behaviourally-equivalent. Trace-equivalence is less useful once the restriction on deterministic processes or on all input being provided at once is removed.

Definition 2.2.4. Restriction Semantics and Relabelling
Restrictions and relabellings hold the property:

$$P \xrightarrow{ax} P' \implies \begin{cases} P\backslash b \xrightarrow{ax} P'\backslash b & \text{if } a \notin \{b, \bar{b}\} \\ P[S] \xrightarrow{S(a)x} P'[S] \end{cases}$$

That is, a process is equivalent under renaming if any actions on that process are also renamed.

Definition 2.2.5. Identifier Semantics
Suppose a behaviour identifier b is defined (possibly recursively) as $b(x_1 \dots x_n) \iff P$, and that for the process P, $FV(P) \subseteq \{x_1 \dots x_n\}$. Then processes may be reduced as follows:

$$P[v_1/x_1 \dots v_n/x_n] \xrightarrow{ax} P' \implies b(v_1 \dots v_n) \xrightarrow{ax} P'$$

The identifier operation can be seen as similar to abstractions ($\lambda x.M$) in the λ-calculus.

Remarks. CCS excels in providing a powerful language for describing high-level concurrent systems. Note the limit on only inputting and outputting variables and expressions, as well as the asynchronous nature of inter-process communication. However it struggles to describe the low-level atomic actions. An encoding of, say, a list is difficult as the language revolves around systems communicating with one another through input/output synchronisation.

The Tea/Coffee Problem should be kept in mind for the following calculi, particularly for the Solo calculus.

2.3 π-calculus

The π-calculus was designed by Robin Milner, Joachim Parrow and David Walker in 1992 as an extension of Milner's work on CCS. It was supposed to remain similar to the λ-calculus as described in 2.1 and improve on CCS by allowing channel names themselves to be sent across channels.

Definition 2.3.1. Syntax
Explained here by Parrow (2001)*, the π-calculus is constructed from the recursive definition of an expression P:

P	:=	$Q \mid R$	(concurrency)
		$Q + P$	(choice)
		$c(x).Q$	(input prefix)
		$\bar{c}(x).Q$	(output prefix)
		$\tau.P$	(silent prefix)
		if $x = y$ then P	(match)
		$vx.Q$	(restriction) †
		$Q!$	(replication) ‡
		$\mathbf{0}$	(null process)

where all operations are as found in CCS. The notable difference is allowing the sending and receiving of channel names over a channel, similar to passing pointers in traditional programs.

*See Milner (1999) for the original definitions and semantics. The following are functionally equivalent and *I personally believe that the further research on the topic has led to improved definitions and descriptions of the subject.*

†This explicitly declares x as local within Q.

‡Replication is defined in theory as $P! := P \mid P!$. However, this causes problems in computation as to how much to replicate and is in fact computed differently.

It is worth noting that the match and choice operators are not strictly needed, but are often included as they are usually required to be defined within the calculus anyway. The lack of the relabelling operator from CCS (written $[b/a]$) may be surprising at first, but the same can be achieved through a definition. That is, $P[a'/a, b'/b]$ can be recreated within the π-calculus by *identifying* $P(x, y)$ with $P[x/a, y/b]$ in a fashion similar to C-style functions. This is simply for the purpose of saving an otherwise unnecessary operator.

> **Example.** Suppose a process P needs to send multiple names $a, b, c \ldots$ to another process Q, but there exist multiple such Qs that are all listening on the same channel. The naive approach may lead to Q_1 receiving a, Q_2 receiving b etc... P can begin by transmitting a private channel name p, then transmitting each $a, b, c \ldots$ on p:
>
> $$(vp)\, \bar{c}p \,.\, \bar{p}a \,.\, \bar{p}b \,.\, \bar{p}c \,.\, P$$
>
> Then Q must prepare to receive a name then $a, b, c \ldots$ along that named channel, binding each to any desired name:
>
> $$c(p) \,.\, p(x) \,.\, p(y) \,.\, p(z) \,.\, Q$$

Note how a generalisation of this could allow P to send a pair of names (x, a) and Q could bind the value a to the name x, allowing for hash-map or set-like behaviour. *In my opinion, the simplicity of this encoding and the expressiveness it may provide is outstanding.*

> **Remarks.** Built mostly on the work of Milner and CCS, the π-calculus demonstrated that a simplification in design can (and usually does) lead to a more expressive language. There is not much else noteworthy except for the increased ease of use of the calculus over CCS — the encoding of any examples in 2.1 is left as an exercise to the reader in either π-calculus or CCS. While it provides the expressiveness required for Turing-completeness, it does not lend itself to understandability nor clarity of the problem encoding when presented as a standalone expression within the calculus, unless the reader is well-acquainted with the subject in advance. Furthermore, larger expressions quickly become unreadable without liberal use of identifiers.

2.4 Fusion Calculus

Björn Victor and Joachim Parrow first designed the Fusion calculus in the late 1990's as an attempt to simplify the π-calculus further still. They reduced the number of binding operators from two (input/output) to one (fusion). This new operation assigned two names to the same value or channel in one of the processes they were in, meaning the calculus has the bizarre property that all input/output actions are symmetric. The reduction in operators meant a reduction in bisimulation congruences, down now from three to just one, which justifies further mention here.

> **Definition 2.4.1.** Syntax
> As defined by Parrow and Victor (1998), the Fusion calculus is composed from *free actions* ranged over by $\alpha \ldots$ and *agents* ranged over by $P, Q \ldots$ as such:

$$
\begin{array}{rcll}
P & := & \mathbf{0} & \text{(null process)} \\
 & & \alpha.P & \text{(action prefix)} \\
 & & P \mid Q & \text{(concurrency)} \\
 & & P + Q & \text{(choice)} \\
 & & (x)\, P & \text{(scope)} \\
 & & & \\
\alpha & := & xy & \text{(input)} \\
 & & \bar{x}y & \text{(output)} \\
 & & \phi & \text{(fusion)}
\end{array}
$$

where x is bound in $(x)P$.

Definition 2.4.2. Structural Congruence
The structural congruence $\equiv$ is the least congruence relation satisfying α-equivalence, associativity, commutativity, $\mathbf{0}$ identity and the following scope laws:

$$
\begin{aligned}
(x)\,\mathbf{0} &\equiv \mathbf{0} \\
(x)\,(y)\,P &\equiv (y)\,(x)\,P \\
(x)\,(P \mid Q) &\equiv P \mid (x)Q \quad \text{where } z \notin fn(P) \\
(x)(P + Q) &\equiv (x)P + (x)Q
\end{aligned}
$$

This aims to provide the core definition of equivalent computation and a basis for reductions.

Definition 2.4.3. Semantics
Similar to CCS, a reduction in the Fusion calculus is a labelled transition $P \xrightarrow{\alpha} Q$ where: where each satisfies the expression:

$$
\begin{aligned}
\alpha.P &\xrightarrow{\alpha} P \\
P \xrightarrow{ux} P', Q \xrightarrow{\bar{u}y} Q' &\implies P \mid Q \xrightarrow{x=y} P' \mid Q' \\
P \xrightarrow{\alpha} P' &\implies \begin{cases} P \mid Q \xrightarrow{\alpha} P' \mid Q \\ P + Q \xrightarrow{\alpha} P' \end{cases} \\
P \xrightarrow{\phi} P', z\phi x, z \neq x &\implies (x)P \xrightarrow{\phi \backslash z} P'x/z \\
P \xrightarrow{\alpha} P', x \notin names(\alpha) &\implies (x)P \xrightarrow{\alpha} (x)P' \\
P \equiv P', Q \equiv Q', P \xrightarrow{\alpha} Q &\implies P \xrightarrow{\alpha} Q \\
P \xrightarrow{(y)ax} P', z \in \{x, y\}, a \notin \{z, \bar{z}\} &\implies (z)P \xrightarrow{(zy)ax} P'
\end{aligned}
$$

Note that all of the above can be easily generalised for action objects being k-length tuples rather than single values.

Example. Within the Fusion calculus — like the π-calculus — input and output actions block computation of a process until the input/output action is performed. However, we may define an asynchronous, delayed input that allows P to continue with computation before binding an inputted value x from a channel u.

$$u(x) : P := (x)(ux \mid P)$$

Remarks. There exists an encoding of the π-calculus within the Fusion calculus. This is not so interesting at this point but is later.
The Fusion calculus shows that simplicity is definitely desirable in a process calculus as the reduction of rules and syntax eases readability and aids in proving properties of expressions. Despite this, *I believe it still does not express suitably well the function of an expression without applying much thought or computation.*

2.5 Solo Calculus

Developed by Cosimo Laneve and Björn Victor in the early 2000s, the Solo calculus aims to be an improvement of the Fusion calculus. As such, there exists an encoding of the Fusion calculus within the Solo calculus (and hence an encoding of the π-calculus). The name comes from the strong distinction between the components of the calculus: *solos* and *agents*. These are roughly analogous to input/output actions and a calculus syntax similar to the λ-calculus. Through some clever design choices, the Solo calculus is found to have some interesting properties over other process calculi.

Definition 2.5.1. Syntax
As defined by Laneve and Victor (1999), the Solo calculus is constructed from *solos* ranged over by $\alpha, \beta \ldots$ and *agents* ranged over by $P, Q \ldots$ as such:

$$
\begin{array}{lcll}
\alpha & := & u\,\tilde{x} & \text{(input)} \\
 & & \bar{u}\,\tilde{x} & \text{(output)}\ ^{*} \\
 & & & \\
P & := & 0 & \text{(inaction)} \\
 & & \alpha & \text{(solo)} \\
 & & Q \mid R & \text{(composition)} \\
 & & (x)\,Q & \text{(scope)} \\
 & & !\,P & \text{(replication)}
\end{array}
$$

where the scope operator $(x)\,P$ is a declaration of the named variable x in P. This ensures that x is local to P, even if it assigned outside of P (ex. $(x\,y|(x)\,P)$ will never have $x := y$ unless explicitly assigned such in P).

This is a much more minimal syntax when compared to CCS (and certainly Higher-Order CCS as described by Xu (2009)). It will further be seen that the reduction rules retain this simplicity. It should be noted that the names $u, x, etc \ldots$ within a solo may be treated as both channel names and as values.

Remark. Match Operator
The full definition by Laneve and Victor (1999) also includes the match operator $[x = y]\,P$ which computes P if x and y are the same name, otherwise computes $\mathbf{0}$. It has been shown that the inclusion of the match operator is in fact extraneous, so it is excluded.

Definition 2.5.2. Structural Congruence
The structural congruence relation $\equiv$ in the Solo calculus is exactly that defined in Definition 2.4.2.

[*] $\tilde{x}$ is used as shorthand for any tuple $(x_1 \ldots x_n)$.

Definition 2.5.3. Reduction
Reduction semantics on solo expressions are defined as:

$$(x)(\bar{u}\,x \mid u\,y \mid P) \to P\{y/x\}$$

$$P \to P' \implies \begin{cases} P \mid Q \to P' \mid Q \\ (x)\,P \to (x)\,P' \\ P \equiv Q \text{ and } P' \equiv Q' \implies Q \to Q' \end{cases}$$

where $P\{y/x\}$ is α-substitution of the name x to the name y.

It is interesting to note here the asynchronous behaviour of the Solo calculus. Where in the π-calculus and CCS input/output actions where synchronised and preceded processes as guards, the Solo calculus naturally treats all agents as unguarded and names may be substituted whenever is desired. That is, there is no sequential aspect to the calculus, allowing for useful structural congruences.

Remark. There exists an encoding of the Fusion calculus within the Solo calculus. This can most easily be seen as an encoding of the choice-free Fusion calculus as a combination of the above syntax and semantics of the Solo calculus and also the prefix operator $\alpha\,.\,P^*$. Hence there exists an encoding of the π-calculus also, complete with the same style of guarded input/output communication.

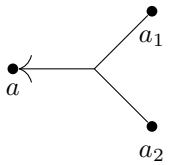
2.6 Solo Diagrams

The Solo calculus was further developed by Laneve et al. (2001) to provide a one-to-one correspondence between these expressions and 'diagram-like' objects. This provides a strong analog to real-world systems and an applicability to be used as a modelling tool for groups of communicating systems. Furthermore, as discussed by Graf et al. (2008), a visual output of information is often found to be preferable for cognition than verbal or textual information.

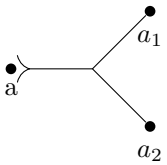
Definition 2.6.1. Edge
An edge is defined to be:

$$E := \langle a, a_1 \dots a_k \rangle_t \quad \text{for } t \in \{i, o\}$$

where a, a_i are *nodes*, $\langle \dots angl_i$ is an *input edge*, $\langle \dots angl_o$ is an *output edge* and k the edge's *arity*.



Output edge $\langle a, a_1, a_2 \rangle_o$



Input edge $\langle a, a_1, a_2 \rangle_i$

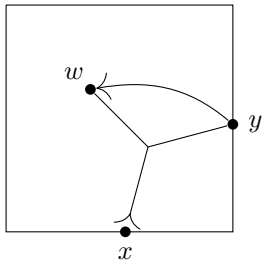
This is analogous to an input or output solo in the calculus, where a is u or $\bar{u}$ and $a_1 \dots a_n$ is $\tilde{x}$ as written in Definition 2.5.1. Note that inputs and outputs must have matching arity — a 2-arity input cannot communicate with a 3-arity output for obvious reasons.

*For further details, Laneve and Victor (1999) discuss this implementation in Section 3

Definition 2.6.2. Box
A box is defined to be:

$$B := \langle G, S \rangle \quad \text{for } S \subset nodes(G)^*$$

where G is a *graph* (or multiset of *edges*) and S is a set of *nodes*, referred to as the *internal nodes* of B. The *principal nodes* of B are then $nodes(G) \setminus S$.



†

Box representing $!(w)(x\,wy \,|\, \tilde{w}\,y)$

This can then be seen to be analogous to the replication operator, with the idea being that the principal nodes form the perimeter of a box and cannot be replicated — they serve as the interface to the internals of the box.

Definition 2.6.3. Diagram
A solo diagram is defined to be:

$$SD := (G, M, \ell)$$

where G is a finite multiset of *edges*, M is a finite multiset of *boxes* and ℓ a labelling of the $nodes(G)$ and of $principals(M)$.

From here, we can convert Solo calculus to diagrams, where composition is intuitively just including two separate diagrams together and scope is simply any connected nodes labelled by ℓ. There are then four required reduction cases (edge-edge, edge-box, box-box and box internals) which can be deduced from the definition of the calculus.

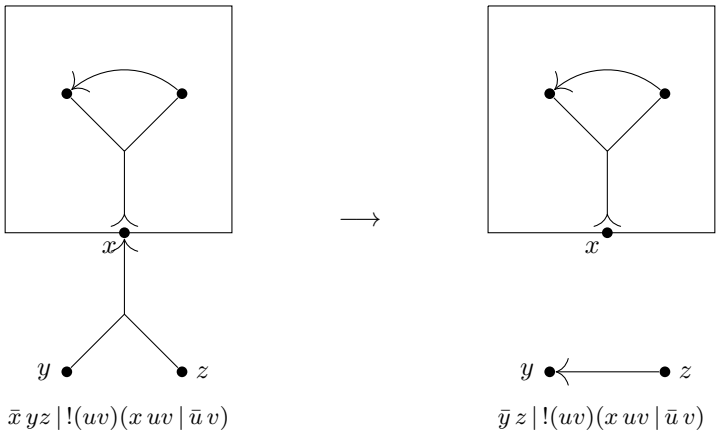
Definition 2.6.4. Diagram Reduction
Let $G, G_1, G_2 \ldots$ be arbitrary graphs, M, M' arbitrary box multisets, $\alpha := \langle a, a_1 \ldots a_k \rangle_i$, $\beta := \langle a, a'_1 \ldots a'_k \rangle_o$, $\sigma := a_i \mapsto a'_i$, ρ a arbitrary but fresh relabelling and $G\sigma$ shorthand for $G[\sigma]$ the application of the renaming σ on the edges of G. α and β need not be fixed to input and output respectively, but must be opposite polarity. Then, the following reductions may be made:

$$(G \cup \{\alpha, \beta\}, M, \ell) \to (G\sigma, M\sigma, \ell')$$
$$(G_1 \cup \{\alpha\}, M := \langle G_2 \cup \{\beta\}, S \rangle, \ell) \to ((G_1 \cup G_2\rho)\sigma, M\sigma, \ell')$$
$$(G, M := \{\langle \{\alpha\} \cup G_1, S_1 \rangle, \langle \{\beta\} \cup G_2, S_2 \rangle\} \cup M', \ell) \to ((G \cup G_1\rho \cup G_2\rho)\sigma, M\sigma, \ell')$$
$$(G, M := \langle \{\alpha, \beta\} \cup G_1, S \rangle \cup M', \ell) \to ((G \cup G_1\rho)\sigma, M\sigma, \ell')$$

*This is written as shorthand for all nodes contained within a given object, in this case $\{a$ s.t. $a \in nodes(S),\ S \in G\}$

†Usually the w in the diagram would be excluded, but is included here for illustration purposes only.

where each represents reduction of an edge-edge, edge-box, box-box and of box internals respectively.



Example.

Remark. The graph objects associated with Solo diagrams are particularly similar to bigraphs. These are a combination of a 'link graph' and a 'place graph', where the collection of nodes and edges forms the former and the subgraphs within boxes dictate the latter. Despite these two similar areas of research, including overlapping research by Milner (2006), it is found to be less relevant for this topic.

The Solo calculus is found to be simple, expressive and remarkable in its capability to be visualised as a diagram. For further reading, Ehrhard and Laurent (2010) present in great detail the topics of the π- and Solo calculus, Solo diagrams and furthermore differential interaction nets.

3 Technology

3.1 Graph Visualisation

The system as a whole is planned to be implemented in the Python language, chosen for its ease of use and ability to be written in both a functional and object-oriented style, with each style likely suited to calculus and diagram implementation respectively. This being said, Python is seen as likely unsuitable for diagram visualisation and as such other solutions are mentioned here.

Definition. Nice Diagrams
For the output of diagrams by a system, it is seen as desirable to create 'nice diagrams'. Some obvious requirements are:

- Fills the given space evenly

- Minimal overlap of edges and of nodes
- Lengths of edges are consistent
- Diagram will adapt as the graph changes

Here, 'nice diagrams' is found to be mostly equivalent to springy force-directed diagrams such that addition or subtraction of parts of the graph, or manual manipulation of position, still leaves a graph that has nodes distributed evenly.

Examples. The implementation of a solution to such a problem is nontrivial and, in the general case, performance-intensive, so is seen outside of the scope of this project. Instead, there exist several software libraries capable of producing these kind of outputs. Notable mentions include:

- *GraphViz* for C *
- *igraph* for C *
- *springy.js* for ECMAScript
- *VivaGraph.js* for ECMAScript
- *d3.js* for ECMAScript

Remarks. Most of these libraries, especially those that allow interactivity with the output, are written for web-browsers and are subsequently written in ECMAScript as the popular choice. With this in mind, this project is likely to focus on the use of the latter-mentioned *d3.js*, which produces an interactive output or SVG from a JSON source. The reasoning for this choice is based upon both the features available and the relative maturity of the library. The popularity of the library across a large number of people, coupled with a development history dating back to early 2011, provides evidence of a usable, feature-rich solution. Furthermore, the open-endedness of program I/O through use of HTTP GET/PUT/POST ensures language interoperability will not be hindered too much.

4 Solo Calculus

The implementation here was divided into two parts: a collection of objects and functions and a read-eval-print-loop (REPL) interface. The former handles reduction strategies, string formatting of terms and the underlying data structure. The latter handles user input and output, wrapping the objects and functions in a human-useable manner.

4.1 Calculus Implementation

4.1.1 Overview and Analysis

The implementation of reduction semantics and calculus objects was achieved through converting all expressions to a canonical normal form. First, we must gather some possible structural congruences.

*These both have many language-specific APIs, so can be used from multiple languages and environments. The original library and interface is however written in C.

Lemma 4.1.1. Structural Equivalence
For any agents $P, Q, \ldots$, the following structural equivalences hold.

$$\begin{aligned}
(x)(y)P &\equiv (x\,y)P \\
P \mid (Q \mid R) &\equiv P \mid Q \mid R \\
P \mid (x)Q &\equiv (z)(P \mid Q\{z/x\}) \quad \text{where} \quad z \notin names(P) \cup names(Q) \\
!(x)(P \mid !Q) &\equiv (z)(!(x)(P \mid z\,\tilde{x}) \mid !(\tilde{w})(\bar{z}\,\tilde{w} \mid Q\{\tilde{w}/\tilde{x}\})) \quad \text{where} \quad \tilde{x} := fn(P)
\end{aligned}$$

Using these congruences, we may reorder terms of expressions through α-equivalence only.

Definition 4.1.2. Normal Form
An agent P is of normal form *iff*

$$\begin{aligned}
P &\equiv (\tilde{x})(\prod_i y_i\,\tilde{z_i} \mid \prod_j !\,Q_j) \\
\text{where} \quad Q_j &\equiv (\tilde{x})(\prod_k y_k\,\tilde{z_k})
\end{aligned}$$

That is, P is written as a 3-tuple of a scope, a composition of solos and a composition of replicators, where each replicator is written as a 2-tuple of a scope and a composition of solos only.

Lemma 4.1.3.
For any agent P, $\exists\, Q \equiv P$ such that Q is in normal form.

Proof. The proof is trivial by applying recursively the equivalences above. See below for an algorithm for such a construction. □

Algorithm 1 Construction of Normal Forms

Input: Agent P a Normal Form, Q non-Normal Form
Output: Agent P', Normal Form of P and Q

```
1:  function NORMALISE(P)
2:      if Q a Scope then
3:          (b̃n)Q' := Q
4:          collisions := b̃n ∩ names(P)
5:          if collisions ≠ ∅ then
6:              α : collisions → fresh names
7:              return NORMALISE(P, α(Q))
8:          else
9:              return (b̃n)NORMALISE(P, Q')
10:         end if
11:
12:     else if Q a Composition then
13:         Q_1 | ... | Q_n := Q
14:         return REDUCE(Normalise, P, Q_1 ... Q_n)
15:
```

```
16:     else if Q a Replication then
17:         Q' := NORMALISE(Q'') where !Q'' := Q
18:         if replicators(Q') ≠ ∅ then
19:             Q̄ := FLATTEN(Q)
20:             return NORMALISE(P, Q̄)
21:         else
22:             collisions := bn(P) ∩ bn(Q')
23:             α : collisions → fresh names
24:             return P | !α(Q')
25:         end if
26:
27:     else if Q a Solo then
28:         (b̃n)P' := P
29:         collisions := b̃n ∩ names(Q)
30:         if collisions ≠ ∅ then
31:             α : collisions → fresh names
32:             return NORMALISE(α(P), Q)
33:         else
34:             return (b̃n)(P' | Q)
35:         end if
36:     end if
37: end function
```

Normalisation is then performed on an agent P by `Normalise`(P) := `Normalise`$(0, P)$ It will be seen later that this is exactly equivalent to how Solo diagrams are represented. From here onwards, it is assumed that any agent has already been converted to normal form where appropriate.

The algorithm for reductions is a simple search problem for two solos of matching subject and arity (number of names) and opposite parity (input vs. output).

Algorithm 2 Reduction of Solos

Input: Agent P
Output: Agent P', a reduction of P

```
1: function REDUCE(P)
2:     for each i ∈ inputs(P) do
3:         for each o ∈ outputs(P) do
4:             if i agrees with o then
5:                 σ := FUSE(i, o, bn(P))
6:                 if σ ≠ none then
7:                     return σ(P − {i, o})
8:                 end if
9:             end if
10:        end for
11:    end for
12:    return P
```

13: **end function**

The next step is to find a suitable σ, should one exist. This is a renaming of the names of P and must satisfy:

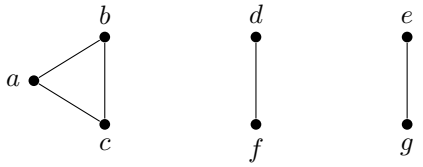
$$\sigma : bn(P) \to names(P)$$
$$\emptyset = domain(\sigma) \cap range(\sigma)$$
$$\sigma(x) = y \implies x \text{ and } y \text{ have been fused}$$

This is found through converting the list of pairs of object names to edges of a graph. For each disconnected subgraph, or partition, the span of names forms a set which must all be fused into one another. Subsequently, there must be at most one free name in this set as two or more would require renaming one free name to another, which is not allowed. If no free name exists, a bound name is chosen as the 'free name' at random. σ is then constructed by $\sigma(bn) := fn$.

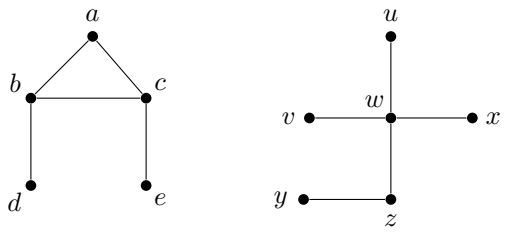
Algorithm 3 Fusion of Solos

Input: $i_1 \dots i_n$ objects of solo i, $o_1 \dots o_n$ objects of solo o, bn set of bound names
Output: $\sigma : bn \to names(i) \cup names(o)$ or *none*

1: **function** FUSE($i_1 \dots i_n$, $o_1 \dots o_n$, bn)
2: `Graph` $g := \{(i_j, o_j) \mid 1 \leq j \leq n\}$
3: `Map` $\sigma := id$
4: **for each** `Graph` $\bar{g} \in partitions(g)$ **do**
5: $isect := nodes(\bar{g}) - bn$
6: **if** $|isect| = 0$ **then**
7: $fn := x \in nodes(\bar{g})$
8: **else if** $|isect| = 1$ **then**
9: $fn := x \in isect$
10: **else**
11: **return** *none*
12: **end if**
13: **for each** $name \in nodes(\bar{g}) - \{fn\}$ **do**
14: $\sigma(name) := fn$
15: **end for**
16: **end for**
17: **return** σ
18: **end function**



$\texttt{Fuse}((a, b, c, d, e),\ (b, c, a, f, g)) \to \{\{a, b, c\}, \{d, f\}, \{e, g\}\}$



$\texttt{Fuse}((a, a, u, b, b, c, v, w, w, y),\ (b, c, w, c, d, e, w, x, z, z)) \to \{\{a, b, c, d, e\}, \{u, v, w, x, y, z\}\}$

Examples.

This forms a complete implementation of non-replicating parts of the calculus. To avoid the implementation problems with replicators mentioned in Laneve et al. (2001), it is necessary to only perform expansions on replicators that can be reduced. While the method described in the aforementioned paper would be suitable, it is enough to simply search for replicators which may be reduced, but only expand them rather than perform a complete reduction. This eases some implementation details and minimises code duplication while remaining correct. The expansion will be reduced on the next pass of the `Reduce` function. Below shows an extension of the `Reduce` function for a replicator-replicator fusion (reduction of the form $!P \mid !Q$).

Algorithm 4 Reduction of Replicators

Input: Agent P
Output: Agent P', a reduction of P

```
1:  function REDUCE(P)
2:      for each i ∈ inputs(P) do
3:          ...
4:      end for
5:
6:      for each i ∈ inputs(R_i) | R_i ∈ ⋃_{!R∈P} R do
7:          for each o ∈ outputs(R_o) | R_o ∈ ⋃_{!R∈P} R do
8:              if i agrees with o then
9:                  σ := FUSE(i, o, bn(P) ∪ bn(R_i) ∪ bn(R_o))
10:                 if σ ≠ none then
11:                     return P ∪ {R_i | R_o}
12:                 end if
```

```
            end if
        end for
    end for
    return P
end function
```

The third case of solo-replicator fusion (reduction of the form $P \mid !Q$) is then trivial.

4.1.2 Testing and Correctness

Testing of this section was integrated with testing of the REPL by using the helper functions it provided for constructing agents. An in-depth description can be found in 4.2.2.

Early tests could be tested by string equalities and this was sufficient. However, more complicated agents led to non-deterministic results and non-trivial equalities. The first example of this occured while comparing the following agents:

$$(x)p\,x \quad \equiv \quad (y)p\,y \quad \neq \quad (x)p\,y$$

Before testing could be achieved in an automated manner, an algorithm for testing for α-equivalence was first required.

It will be seen later that conversion of calculus expressions into diagrams intuitively shows α-equivalence is reducible to graph isomorphism, so is expected to be of poor performance.

Algorithm 5 α-Equivalence of Agents

Input: Agents P, Q
Output: Map $\alpha : names(P) \to names(Q)$ *s.t.* $\alpha(P) \equiv Q$ or *none*

```
function α-Equivalence(P, Q)
    if P ≡ Q then
        return id
    end if
    for comb_scope ∈ combinations(scope(P), scope(Q)) do
        α_{P,Q} : x ↦ y ∀ (x, y) ∈ comb_scope
        for comb_replicator ∈ combinations({R | !R ∈ P}, {R | !R ∈ Q}) do
            for each (r_P, r_Q) ∈ comb_replicator do
                α_{r_P,r_Q} := α-Equivalence(r_P, r_Q)
            end for
            if none ∈ {α_{r_P,r_Q} | (r_P, r_Q) ∈ comb_replicator} then
                continue for
            else if ∃ α_1, α_2 ∈ {α_{r_P,r_Q}} s.t. α_1 disagrees with α_2 then
                continue for
            else
                α := α_{P,Q} ∪ ⋃_{α∈{α_{r_P,r_Q}}} α
            end if
            if α(P) ≡ α(Q) then
                return α
```

```
            end if
        end for
        if α(P) ≡ α(Q) then
            return α
        end if
    end for
    return none
end function
```

4.1.3 Discussion

The initial implementation stayed closer to the core concepts of the calculus. There existed separate classes of objects for each of Scopes, Compositions, Replications, Matches and Solos. While such an approach is perhaps more intuitive, problems arise due to conflicts between how the data structure naturally appears to be tree-like where properties of agents are functions of themselves and their children only and how whether or not a name is bound or free is a property dependent upon an agent or its parent, grandparent etc.

Example. Consider the following equivalent expressions.

$$\begin{aligned} P &:= (y)(a\,x \mid (\bar{a}\,y \mid p\,y)) \\ Q &:= (a\,x \mid (y)(\bar{a}\,y \mid p\,y)) \end{aligned}$$

Both P and Q have the property that y is a bound name and both should reduce to the term $p\,x$. In both cases, y and x are fused into the free name x. The scope of y disappears and the remaining 'print' term displays the result of the fusion. Consider now the (inequivalent) expression.

$$R := (a\,x \mid (y)\bar{a}\,y \mid p\,y)$$

This term, while similar to P and Q, reduces to the term $p\,y$, as our scoped y that is fused is not the same as the free y that is 'printed'. This can be seen through α-equivalence and renaming $(y)\bar{a}\,y$ to the equivalent term $(z)\bar{a}\,z$ and observing that a is a fusion on x and z.
The distinction between the two behaviours is not intuitive, especially in non-trivial cases where agents may be deeply nested.

The above example is further amplified when manipulating expressions symbolically. When searching for reducible patterns, the expression is divided and recombined, rearranging the expression tree and subsequently each node's parents. While the problem is solvable through updating which node has which parent as the expression is rearranged and manipulated, the solution eventually becomes an attempt to form a normal form out of the expression, leading to the revised method described in 4.1.1.

Through the implementation of the calculus, many interesting properties became apparent, with some of the most noticeable listed below.

Lemma. Non-Finite Reductions
There exist expressions for which reductions may be performed indefinitely.

For a reduction to be non-finite, both solos must be replicable. In particular, expressions of the form $(!(\tilde{x})(u\,\tilde{x} \mid P) \mid !(\bar{u}\,\tilde{y} \mid Q) \mid R)$ are non-finite.

Example.
Consider the following expressions, each a fusion of a bound x and free y:

$$P \to P'\{y/x\} \text{ where } x \in !(x)Q \in P \text{ and } y \in !R \in P$$
$$(x)P \to P'\{y/x\} \text{ where } x \in !Q \in P \text{ and } y \in !R \in P$$

The first expression may be reduced indefinitely, each time adding the expansion of Q to P. The second may be performed once, after which each x in Q is renamed to y. It is then ambiguous as to whether it may be performed again — on one hand the fusion is on two free names, on the other the two names are the same, so there is only one distinct free name.

Remark. Behaviour of Trivial Reductions
For this body of work, since the reduction requires $\sigma : \emptyset \to \emptyset$, it was assumed that under such conditions a reduction should not be seen to have happened.

4.2 Read-Eval-Print-Loop

4.2.1 Overview and Analysis

A simple REPL interface was constructed to interact with the code written in 4.1. This was done using extended regular expressions similar to those found in the Perl programming language. These do not meet the normal definitions of regular expressions as they additionally support recursive matches, backreferences and various other syntactic additions that make them context-free grammars rather than regular expressions. In fact, given a suitable engine for applying context-free grammars, this would be a preferable way to parse expressions.

For each agent, there was an extended regex matching its string representation, with backreferences to extract the necessary data. Each of $(?P\langle name\rangle expression)$ represents a named backreference to the given expression and $(?\,\&rec)$ is a recursive call to the outer $(?\langle rec\rangle expression)$ expression.

Definition 4.2.1. Solo Matching
Solos are built using the regex as follows:

$$u\,\tilde{x} \quad \longleftarrow \quad \backslash s?(?P\langle subject\rangle[a-z\,0-9]+)\backslash s(?P\langle objects\rangle([a-z\,0-9]+\backslash s?)+)\backslash s?$$
$$\bar{u}\,\tilde{x} \quad \longleftarrow \quad \backslash s?\backslash\hat{}\,(?P\langle subject\rangle[a-z\,0-9]+)\backslash s(?P\langle objects\rangle([a-z\,0-9]+\backslash s?)+)\backslash s?$$

where a valid name is any lowercase alphanumeric word and $\bar{u}$ is inputted as `^u` and each of `subject` and `objects` represent the subject and objects of the solo respectively.

Definition 4.2.2. Replication
Replicators are built using the regex as follows:

$$!P \quad \longleftarrow \quad \backslash s?\ !(?P\langle agent\rangle.*)\backslash s?$$

where `agent` represents P for the expression $!P$.

Definition 4.2.3. Scope
Scopes are built using the regex as follows:

$$(x)P \quad \longleftarrow \quad \backslash s?\backslash((?P\langle bindings\rangle([a-z\,0-9]+\backslash s?)+)\backslash)(?P\langle agent\rangle[\hat{}\backslash s].+)\backslash s?$$

where each of `bindings` and `agent` represent x and P respectively for the expression $(x)P$.

Definition 4.2.4. Composition
Compositions are built using the recursive regex as follows:

$$\begin{aligned} P_1 \,|\, \ldots \,|\, P_n \quad &\longleftarrow \quad \backslash s?\backslash(?\langle agents\rangle\ \texttt{R}+)(\backslash|(?\&agents))?)\backslash)\backslash s? \\ \text{where} \quad \texttt{R} \quad &:= \quad (?\langle agent\rangle([\hat{}|()]|(?\langle rec\rangle\backslash((?:[\hat{}()]++|(?\&rec))*\backslash))) \end{aligned}$$

where each `agent` is collected in `agents` and represents each of $P_1 \ldots P_n$.

It is accepted that this is difficult to read and would benefit from a better-structured parsing system. However, for this project and as a write-once solution, it is suitable enough.

4.2.2 Testing and Correctness

Testing was first done of the structural equivalence and equality of expressions. First, for each of the congruences described above, a normalised expression and its known normal form were tested for equality. Testing of reductions was done through testing each of the four cases of fusions being performed:

- Normal fusion $(\tilde{x})(a\tilde{x} \,|\, \bar{a}\,\tilde{y})$
- Cross-replicator fusion $(\tilde{x})(a\,\tilde{x} \,|\, !(\bar{a}\,\tilde{y}))$
- Multi-replicator fusion $(\tilde{x})(!(a\,\tilde{x}) \,|\, !(\bar{a}\,\tilde{y}))$
- Inter-replicator fusion $(\tilde{x})(!(a\,\tilde{x} \,|\, \bar{a}\,\tilde{y}))$

where for each, there exist approximately four cases of which variables are bound and where the matching scope lies.
Due to the open-endedness of the problem, it is difficult to provide a conclusive, complete testing suite. In particular, checking for false positives presents a large space of possible tests. This was managed by reducing to the above unit tests and combining with the integration test of each case of fusion.

A sample of test outputs can be found in 8.1

4.2.3 Discussion

No research was done as to the usability of this interface. Future work would vastly improve the description provided here, particularly with regards to error-reporting. Another notable flaw is a disconnect between text inputted and outputted, for example $\hat{}p\ a\ b\ c \to \bar{p}\ a\ b\ c$.

With further work on the project, I would have hoped to embed this REPL interface in the REST server described in 5.2. Through this, the diagram presented by the server could be changed on the server-side.

5 Solo Diagrams

As with the calculus, a suite of functions and objects were written implementing the underlying structure and operations on Solo diagrams. A REST server then presents a computer-useable interface for reduction of diagrams. The REST server alone is of little use, but combined with the diagram visualiser completes the system and allows for visualising diagrams as interactive graphs through a web-browser.

5.1 Diagram Implementation

5.1.1 Overview and Analysis

The implementation of diagrams was done through the definitions provided in Laneve et al. (2001). Beginning with the definition of a complete diagram, an associated class was written for each object and structure using the definitions as written in 2.6. There was one small change to these definitions, where a diagram did not have an associated labelling l. This was instead implemented within the nodes themselves, which were uniquely identified by a UUID4 string but also had a separate (possibly *none*) `name` property. This change meant graphs could have equivalently-named nodes that were functionally distinct.

First, a quick recap of the revised definitions.

Definition 5.1.1. Multiset
A multiset M is an unordered collection of elements $x_1 \dots x_n$. The elements X_i need not be distinct. In particular, M is a map relating each element of the underlying set to a positive integer.

Definition 5.1.2. Diagram
A diagram D is a 2-tuple (G, M) where G is a graph and M a multiset of boxes.

Definition 5.1.3. Graph
A graph G is a multiset of edges.

Definition 5.1.4. Box
A box B is a 2-tuple (G, S) where G is a graph and $S \subset nodes(G)$ set of internal nodes. S is the internal nodes of B and $nodes(G) \setminus S$ the principal or perimeter nodes.

Definition 5.1.5. Edge
An edge E is a k-tuple of names $(x, [x_1, \dots, x_{k-1}])$ and a parity $p \in \{in, out\}$ where x is the subject of E and $x_1, \dots, x_{k-1}$ the objects. E is then written $< x, x_1, \cdots, x_{k-1} >_p$ Note the case $k = 1$ where E has no objects.

Definition 5.1.6. Node
A node x is a unique identifier `uuid` and optionally a labelling `name`.

The implementation of these definitions as a data structure is trivial.

The implementation could end here as there exists an isomorphism between diagrams and calculus expressions as follows:

Lemma 5.1.7. Isomorphism of Diagrams and Expressions

For any diagram $D := (G, M := (G', S))$, there exists a calculus expression P.
In particular, this P is unique up to normal form.

$$P := (\{x \in names(D) \mid x.\texttt{name} = none\})(\bigcup_{e \in edges(D \setminus M)} e \quad | \quad \bigcup_{b \in M} !Q \text{ where } Q \equiv (G', \emptyset))$$

As it can be seen, this is exactly the normal form defined in 4.1.1. This bijection between a normal form of calculus expressions and graph-like diagrams leads to a useful result.

Corollary. Hardness of α-Equivalence Problem
The general case of α-equivalence is equivalent to graph-isomorphism.

With this in mind, the poor worst-case performance of the aforementioned α-equivalence algorithm is known to be unavoidable.

Algorithm 6 Fusion of Nodes

Input: e_i, e_o edges of matching subject and opposite parity $\in D$ a diagram
Output: $\sigma : labelled(D) := \{x \in nodes(D) \mid x.\texttt{name} = none\} \to nodes(D)$ or $none$

```
 1: function FUSE(e_i, e_o)
 2:     x_1, ..., x_n := objects(e_i)
 3:     y_1, ..., y_n := objects(e_o)
 4:     Graph g := {(x_i, y_i) | 1 ≤ i ≤ n}
 5:     Map σ := id
 6:     for each Graph ḡ ∈ partitions(g) do
 7:         isect := nodes(ḡ) ∩ labelled(D)
 8:         if |isect| = 0 then
 9:             fn := x ∈ nodes(ḡ)
10:         else if |isect| = 1 then
11:             fn := x ∈ isect
12:         else
13:             return none
14:         end if
15:         for each name ∈ nodes(ḡ) − {fn} do
16:             σ(name) := fn
17:         end for
18:     end for
19:     return σ
20: end function
```

From here, reduction of diagrams is just a simple change of naming conventions from 4.1.1. Additionally, the stricter structure means there are no edge-cases to consider and box reductions are performed in full, rather than expanding and fusing non-box edges. Again, the following algorithm describes reduction of two edges outside of any boxes and of two solos both inside boxes. The other cases of reduction are trivial to deduce.

Algorithm 7 Reduction of Graphs

Input: D a diagram
Output: D' a reduction of D

1: **function** REDUCE(D)
2: **for each** $\alpha \in edges(D)$ **do**
3: **for each** $\beta \in \{e \in edges(D) \mid parity(e) \neq parity(\alpha)\}$ **do**
4: **if** α agrees with β **then**
5: $\sigma :=$ FUSE(α, β)
6: **return** $\sigma(D \setminus \{\alpha, \beta\})$
7: **end if**
8: **end for**
9: **end for**
10:
11: **for each** $\alpha \in B_\alpha \in boxes(D)$ **do**
12: **for each** $\beta \in B_\beta \in boxes(D)$ **do**
13: **if** α agrees with β **then**
14: $\rho : internals(B_\alpha \mid B_\beta) \to fresh\ nodes$
15: $\sigma := \rho($FUSE(α, β))
16: $G := graph(D) \mid graph(B_\alpha) \setminus \{\alpha\} \mid graph(B_\beta) \setminus \{\beta\}$
17: $M := boxes(D)$
18: **return** $(\sigma(G), \sigma(M))$
19: **end if**
20: **end for**
21: **end for**
22: **end function**

5.1.2 Testing and Correctness

Due to the more abstract nature of diagram objects when compared to calculus expressions, testing was combined with testing of the visualiser. By default, a diagram with parts allowing for each kind of fusion was produced and examined through each reduction.

5.1.3 Discussion

Similarly to the calculus, there exist diagrams that have a non-finite amount of reductions. The behaviour upon reaching a non-finite reduction was left as-is — the diagram would be left to reduce forever even if there were other reducible parts. This could be changed in future implementations to either not perform non-finite reductions if that same reduction was performed recently, or to select edges for reduction with an element of random choice. As is, while technically correct, there are some reductions which are never made despite being possible.

When later combined with the REST server for computer interaction, a collection of functions were written to convert diagrams to JSON objects. Each of these objects held a unique identifier and a collection of that object's properties. The addition of unique identifiers to objects that didn't necessarily require them allowed for improvements in visualisation to be described later.

The most notable difference between the implementation of diagrams and calculus expressions is that the rigid structure of diagrams leaves fewer edge-cases to deal with. Overall, the implementation was far easier and less prone to bugs, *whether or not this was because of my experience in implementing the calculus prior to the diagrams or because of a superior design.*

5.2 Visualiser and REST Server

5.2.1 Overview and Analysis

Diagrams produced by the above sections were visualised as graphs of nodes and edges within an SVG (scalable vector graphic). Further, this graph was made interactive using ECMAScript's d3.js library, in particular using force-directed graphs. This was coupled through a back-end server acting as a reduction engine with a representational state transfer (REST) architecture and a front-end webpage with ECMAScript components for interacting with the output of the server.

The REST server was a single endpoint with options of GET and POST requests. GET returned an example diagram as a JSON object — in particular, this was a collection of four disconnected diagrams, one for each type of edge reduction. POST would, given a JSON object, reduce and return a new diagram.

> **Remark.** Cross-Origin Resource Sharing (CORS)
> As a note on the bureaucracy of various web standards, the implementation written for this project involved making cross-origin HTTP requests. By default, this behaviour is not allowed, so an additional CORS layer is wrapped around the server to allow such requests.

The front-end visualisation was, in comparison, far more complex. To begin, there would be a GET request to the REST server for the default diagram. The returned JSON object was parsed to an SVG. In particular, SVG groups were made of edges, nodes and boxes. Each class of edge, node and box contained a set of polylines, a circle and a collection of both nodes and edges respectively. Nodes were coloured depending upon the property of their position relative to boxes: external, perimeter or internal.

Interactivity was added using an ECMAScript script and additionally a 'Reduce' button. Each node was made draggable and forces were added to each node and edge. In particular:

- Edges were given spring forces to allow them to stretch on drag actions and contract to a relaxed length otherwise.
- Nodes were given repulsive charges to allow them to spread themselves out evenly over the available space.
- The graph as a whole was given a small central attractive charge to keep the graph centered in the viewing panel.

The 'Reduce' button would then make a POST request to the REST server to update the current graph dataset with its reduction.

Due to the multiple paradigms and flexibility of the language, a small excerpt of code can be found in 8.2. In particular, this utilises a functional programming style — the algorithm amounts to multiple filter-map-reduce operations on the dataset. For example, graph edges (links) are added through the following method:

- Create an SVG group for all edges
- Select all current and new edges
- Create an SVG group for each edge
- Select all sub-edges to each edge (one sub-edge for the subject node, then one for each object node)
- Create a polyline for each sub-edge

The filter operation is only necessary to manage discrepancies in how nodes are partitioned into those in boxes and those not. The naive method would be to gather the sets of nodes referenced in each graph's set of edges. However, principal nodes on the perimeter of boxes will be included both within their respective box and in the surrounding graph (unless the box is disconnected). For this reason, principal nodes are considered part of the parent graph — consider the case of two boxes sharing a principal node, this requires the principal to either be in both boxes or in the parent graph. Furthermore, when black box styles are drawn around diagram boxes, nodes on the perimeter may be outside the box as drawn, but the end of the polyline connecting the node to the internal graph must stay in the drawn box. This leaves perimeter nodes sitting precisely on the perimeter, neither wholly inside or out.

On the event of a reduction being requested, the above creates a new SVG. This leaves new nodes inheriting a default position, visually manifested as a 'jump' of all nodes to the origin $(0, 0)$ in the top-left corner. This is not particularly appealing, so additionally when a new set of nodes and edges is received, positions are copied over where the unique id of nodes match one another. While only a small detail, this change leads to visually 'smooth' reductions and is far more intuitive as to what is happening within the graph.

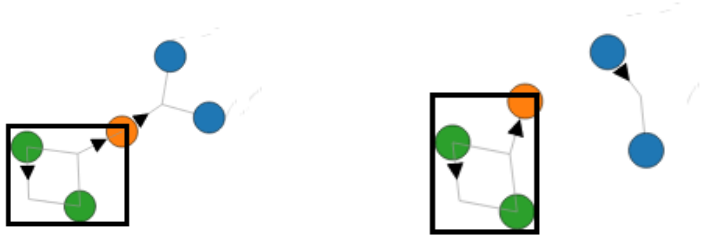
5.2.2 Testing and Correctness

The particularly clear and interactive nature of the diagrams, coupled with the difficulty of constructing an α-equivalence algorithm (graph isomorphism) led to testing being conducted visually. The following exhibits the reduction of some simple diagrams. While the palette of colours used is fixed, the choice of which colour is assigned to which property is not fixed, so colours do change between examples.

Examples.
The first case here shows a normal fusion across two edges. The arrowheads represent the parity of the edge and, in implementation but not shown here, there exist mouseover text boxes with the name of the node if it exists. What should be notable is the potential ambiguity behind this visualisation — the repeated object within an edge is non-obvious aside from a very slight difference in darkness of the line.

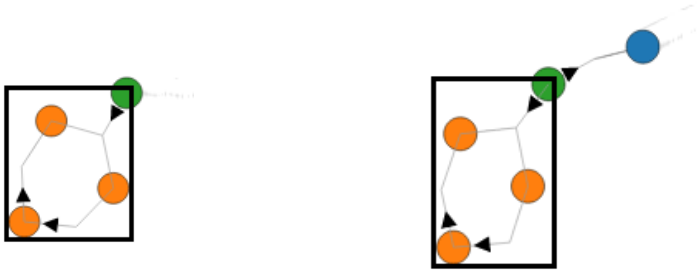
$$(a\,c)(\bar{b}\,a \mid b\,c \mid p\,a\,b\,c) \quad \rightarrow \quad (x)(p\,x\,b\,x)$$

This case demonstrates a reduction between an edge and a box. This exhibits both how perimeter nodes lie exactly on the perimeter, while internal nodes lie entirely inside. The use of colours also makes clear the position of each node relative to the box.



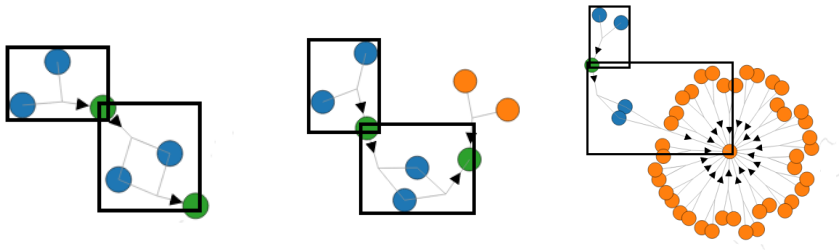
$$(!(x\,y)(\bar{u}\,x\,y \mid x\,y) \mid u\,a\,b) \quad \rightarrow \quad (!(x\,y)(u\,x\,y \mid x\,y) \mid a\,b)$$

Again the diagram demonstrates how the colours change around boxes, but here on a internal box reduction. This, like the next diagram, could be reduced ad-infinitum, but such effect will be shown later. In particular, this demonstrates the explicit creation of new nodes, but again there is ambiguity in the number of objects of the newly created edge (two objects, not one).



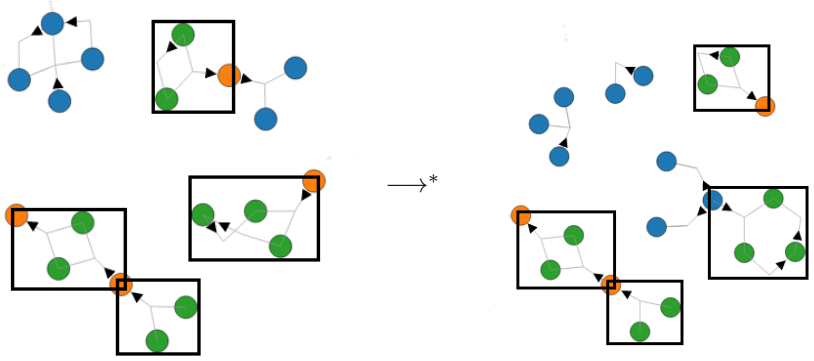
$$!(a\,b)(\bar{x}a \mid x\,b \mid p\,a\,b) \quad \rightarrow \quad (!(a\,b)(\bar{x}a \mid x\,b \mid p\,a\,b) \mid (x)\,p\,x\,x)$$

Completing the set of cases of fusions, an example of a reduction between two boxes, demonstrating the non-finite nature of the reductions. Like the first example, there exists some ambiguity as to the order of objects across a node — there is no obvious way to tell which combination of pairs two edges are matching together. In a larger context the distinction is usually more obvious, but the presence of such ambiguity is recognised to be a shortfall.



$$(!(a\,b)(u\,a\,b) \mid !(c\,d)(\bar{u}\,c\,d \mid w\,c\,d)) \quad \to \quad (!(a\,b)(u\,a\,b) \mid !(c\,d)(\bar{u}\,c\,d \mid w\,c\,d) \mid (x\,y)\,w\,x\,y \mid \ldots)$$

Together, these diagrams may be reduced one after the other. A full chain of step-by-step reductions can be found in 8.3.



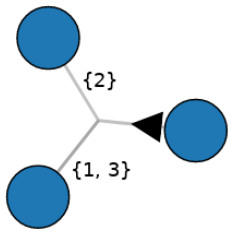
5.2.3 Discussion

While the final example may not seem like a full reduction as the box-box diagram has not reduced, the choice of whether to reduce the internal box example or the box-box is not 'fair'. That is, there is no guarantee that a given reduction will be performed if there exists a non-finite chain of reductions that may be performed instead. This behaviour is not ideal, but isn't necessarily incorrect. Should there be further development, this effect should be addressed early. In the case of reductions involving only boxed edges, alternative behaviour

could be defined to check whether the set of nodes after a reduction differs from the set before a reduction. In this way, while not compromising the multiset properties of diagrams, the pitfalls may be avoided — the reduction may be performed again, but only once the residue of the last reduction has been affected.* Further options would include performing multiple reductions across the diagram as a whole in one step, ensuring that every reduction that can be performed will be performed. However, this will lead to excessive reductions on such non-finite subdiagrams. *I believe this area more than most is still requiring further research and development.*

The multiple cases of ambiguity leave something to be desired. From a UX point of view, the implementation is lacking here, *but I have yet to find a suitable improvement.* Nodes could be discreetly numbered or labeled to describe their order in the list of objects of an edge, however this numbering will differ on an edge-by-edge basis. Numbering or labelling edges would be ideal, however the decisions made in development would make such a feature difficult to introduce. The issues with repeated objects in the same edge could be improved by adding extra shape to the lines drawn. As the line is drawn as a single line segment from the hidden 'connection' node of the edge to the object, adding an extra node between the two, along with the static repulsion of all nodes, would clearly demonstrate nodes with multiple occurrences in a single edge's objects. Alternatively, given a labelling of the index of a node in an edge's objects as suggested formerly, this could further be a set of indices for each occurrence.

Example. Improvements to Labelling of Edges and Objects



6 Conclusion

Over the course of the project, a set of interacting systems have been built. A representation of a calculus expression may be built from a textual input. From here, it may be converted to a normal form, which itself can be trivially converted to a diagram. *With further work, I would like to have seen a connect between the REPL calculus interface and the diagram REST server.* This normal form may then be reduced to a simpler expression, with functions for testing for α-equivalence. Diagrams may be constructed, albeit only through hard-coding them into the application. These diagrams can be reduced and converted to an abstract representation through JSON. Each of these diagrams may then be visualised, manipulated and examined through each reduction step. These visualisations of the diagram

*This is not a perfect solution still as one can build a similar infinite loop between two repeated reductions that interact with one another etc. . .

objects are believed to be clear and allow for a more intuitive understanding of the subject of process calculi.

Through this project's development, *I believe that the diagram objects serve as a superior representation of the calculus.* The mathematical purity to Solo Diagrams is something to be admired. This, combined with the relative lack of further research, leaves a strong case for deeper investigation into Solo Diagrams.

The shortfalls of the project can mostly be attributed to implementation difficulties in the technicalities of the Solo Calculus. However, there could be much improvement on the visualiser, for better differentiating cases of ambiguity and for allowing richer interaction on the user end. *Through further work, I would like to see the ability to build, destroy and edit diagrams from within the visualisation program.*

The specifics of finding general solutions to usually very human-solvable problems such as α-equivalence and construction of suitable σs in reduction steps has given new perspective on the implied vs in-practice hardness of said problems. Contrary to the comment made in 5.1.1 as to reduction of diagrams being done through conversion to calculus expressions, *I believe it would be easier to approach the problem from the opposite direction. Implementation of the Solo Calculus would be eased through converting expressions to diagrams and reducing through their set of rules instead.* However, calculus expressions remain the obvious choice as an interface in situations concerning textual input and output.

Future research and development towards the subject of Proof Nets could prove influential and extensive. The reader is referred to later sections in the work of Laneve et al. (2001) and of Ehrhard and Laurent (2010).

7 References

Dean, Jeffrey and Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. In *Commun. ACM*, vol. 51(1):107–113 (2008). ISSN 0001-0782.

Ehrhard, Thomas and Laurent, Olivier. Acyclic Solos and Differential Interaction Nets. In *Logical Methods in Computer Science*, vol. 6(3):1–36 (2010). ISSN 1860-5974.

Graf, Sabine, Lin, Taiyu and Kinshuk, Taiyu. The relationship between learning styles and cognitive traits Getting additional information for improving student modelling. In *Computers in Human Behavior*, vol. 24(2):122–137 (2008). ISSN 0747-5632.

Hoare, C. A. R. Communicating Sequential Processes. In *Communications of the ACM*, vol. 21(8):666–677 (1978). ISSN 0001-0782.

Laneve, Cosimo, Parrow, Joachim and Victor, Björn. Solo Diagrams. In *Theoretical Aspects of Computer Software: 4th International Symposium, TACS 2001 Sendai, Japan, October 29–31, 2001 Proceedings*, (pp. 127–144). Springer Berlin Heidelberg (2001). ISBN 978-3-540-45500-4.

Laneve, Cosimo and Victor, Björn. Solos in Concert. In *Automata, Languages and Programming: 26th International Colloquium, ICALP'99 Prague, Czech Republic, July 11–15,*

1999 Proceedings, (pp. 513–523). Springer Berlin Heidelberg (1999). ISBN 978-3-540-48523-0.

Machado, Rodrigo. An Introduction to Lambda Calculus and Functional Programming. In *Theoretical Computer Science (WEIT), 2013 2nd Workshop-School on*, (pp. 26–33). IEEE (2013). ISBN 978-1-4799-3057-9.

Milner, Robin. *A Calculus of Communicating Systems*, chap. 5, (pp. 65–83). Lecture Notes in Computer Science, 92 (1980). ISBN 978-3-540-10235-9.

Milner, Robin. *Communicating and mobile systems : the pi-calculus*. Cambridge University Press (1999). ISBN 0521643201.

Milner, Robin. Pervasive Process Calculus. In *Electronic Notes in Theoretical Computer Science*, vol. 162:255–259 (2006). ISSN 1571-0661.

Parrow, J. and Victor, B. The fusion calculus: expressiveness and symmetry in mobile processes. (pp. 176–185). IEEE Publishing (1998). ISBN 0-8186-8506-9. ISSN 1043-6871.

Parrow, Joachim. An Introduction to the π-Calculus. In *Handbook of Process Algebra*, (pp. 479–543). Elsevier Science (2001). ISBN 1-281-03639-0.

Xu, Xian. Expressing First-Order -Calculus in Higher-Order Calculus of Communicating Systems. In *Journal of Computer Science and Technology*, vol. 24(1):122–137 (2009). ISSN 1000-9000.

8 Appendix

8.1 Calculus Testing

Output of command `python src/calculus/tests.py`

```
======================================================================
TestEqualityOperator

 TestEqualityOperator.TestAlphaEquivalence
(x)(p x) == (y)(p y)
(y)(p y) != (x)(p y)

 TestEqualityOperator.TestMultisetProperty
(x)(p x | p x) != (x)(p x)
======================================================================
TestStandardFusion

 TestStandardFusion.TestOneBound
(x)(u x | u y | p x y) -> ()(p y y)

 TestStandardFusion.TestTwoBound
(y x)(u x | u y | p x y) -> (u0)(p u0 u0)

 TestStandardFusion.TestZeroBound
```

```
()(u x | u y | p x y) -> ()(u x | u y | p x y)
==========================================================================
TestFlatteningTheorem

 TestFlatteningTheorem.TestOneBound
!(q)(p x | !(q y)) -> (y0)(!(y1)(q y1 | y0 y1) | !(q)(p x | y0 y))

 TestFlatteningTheorem.TestTwoBound
!(q y)(p x | !(q y)) -> (y0)(!()(q y | y0) | !(q y)(p x | y0))

 TestFlatteningTheorem.TestZeroBound
!(p x | !(q y)) -> (y0)(!(q0 y1)(q0 y1 | y0 q0 y1) | !()(p x | y0 q y))
==========================================================================
TestCrossReplicatorFusion

 TestCrossReplicatorFusion.TestOneBoundInner
()(u x | !(y)(u y | p x y)) -> ()(p x x | !(y)(u y | p x y))

 TestCrossReplicatorFusion.TestOneBoundOuter
(x)(u x | !()(u y | p x y)) -> ()(p y y | !()(u y | p y y))

 TestCrossReplicatorFusion.TestTwoBound
(x)(u x | !(y)(u y | p x y)) -> (u0)(p u0 u0 | !(y)(u y | p u0 y))

 TestCrossReplicatorFusion.TestZeroBound
()(u x | !()(u y | p x y)) -> ()(u x | !()(u y | p x y))
==========================================================================
TestIntraReplicatorFusion

 TestIntraReplicatorFusion.TestOneBoundInner
()(p x y | !(x)(u x | u y)) -> ()(p x y | !(x)(u x | u y))

 TestIntraReplicatorFusion.TestOneBoundOuter
(x)(p x y | !()(u x | u y)) -> ()(p y y | u y | u y | !()(u y | u y))

 TestIntraReplicatorFusion.TestTwoBound
(y x)(p x y | !()(u x | u y)) -> (u0)(p u0 u0 | u u0 | u u0 | !()(u u0 | u u0))

 TestIntraReplicatorFusion.TestZeroBound
()(!()(u x | u y | p x y)) -> ()(!()(u x | u y | p x y))
==========================================================================
TestMultiReplicatorFusion

 TestMultiReplicatorFusion.TestOneBoundInner
()(p x y | !()(u y) | !(x)(u x)) -> ()(p x y | !()(u y) | !(x)(u x))

 TestMultiReplicatorFusion.TestOneBoundOuter
(x)(p x y | !()(u y) | !()(u x)) -> ()(p y y | !()(u y) | !()(u y))

 TestMultiReplicatorFusion.TestTwoBoundOuterInner
(x)(p x y | !(y)(u y) | !()(u x)) -> (u0)(p u0 y | !(y)(u y) | !()(u u0))
```

```
 TestMultiReplicatorFusion.TestTwoBoundOuterOuter
(y x)(p x y | !()(u y) | !()(u x)) -> (u0)(p u0 u0 | !()(u u0) | !()(u u0))

 TestMultiReplicatorFusion.TestZeroBound
()(p x y | !()(u y) | !()(u x)) -> ()(p x y | !()(u y) | !()(u x))
```

8.2 Graph Visualisation Source Sample

Sample output of file `src/visualisation/graph.js`

```
function svgLink(svgElem, dataJoin, filterBy) {
    svgElem.append("g")
        .attr("class", "links")
        .selectAll(".links")
        .data(dataJoin)
        .enter()
            .filter(filterBy)
            .append("g")
            .attr("class", "edge")
            .selectAll("*")
            .data(function(d) { return d; })
            .enter()
                .append("polyline")
                .attr("stroke-width", function(d) { return Math.sqrt(d.value); })
                .each(function(d) {
                    item = d3.select(this).data([d]);
                    if (d["arrow"] != 0) {
                        item.attr("marker-mid", "url(#mid)");
                    }
                });
}

function svgNode(svgElem, dataJoin, filterBy) {
   svgElem.append("g")
        .attr("class", "nodes")
        .selectAll(".nodes")
        .data(dataJoin)
        .enter()
            .filter(filterBy)
            .append("circle")
            .attr("fill", function(d) { return color(d.group); })
            .call(d3.drag()
                .on("start", dragstarted)
                .on("drag", dragged)
                .on("end", dragended));
}
```

8.3 Full Chain of Diagram Reduction

Full reduction of the union of the examples in 5.2.2