Dance of the Solos

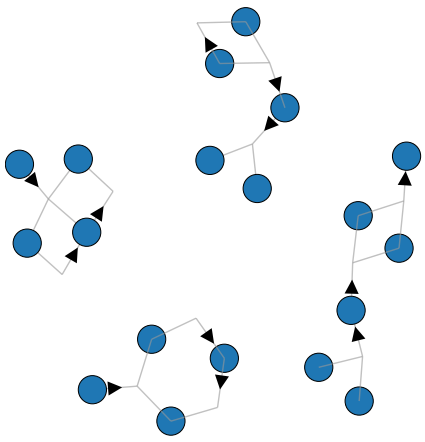
On the graphical representation of Solo Diagrams

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.



Acknowledgements

I would like to express my gratitude to Prof. Guy McCusker for his guidance and dedicated supervision of this project throughout.

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.

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:

$$(\lambda x.P)\,Q \to_\beta P[x ::= Q]$$

$$M \to_\beta M' \implies \begin{cases} P\,M \to_\beta PM' \\ M\,P \to_\beta M'P \\ \lambda x.M \to_\beta \lambda x.M' \end{cases}$$

*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.

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

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

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

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

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

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

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

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 $n ::= 3$:

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

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.*

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

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 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)

*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.

*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 $dom(S) = \{a\}\, ran(S) = \{b\}$

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

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

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:

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

The problem would instead be successfully encoded as:

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

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.

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 $a, b, c \dots$ 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 $a, b, c \dots$ on p:

$$(vp)\, \bar{c}p \,.\, \bar{p}a \,.\, \bar{p}b \,.\, \bar{p}c \,.\, P$$

*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.

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:

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

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:

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

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:

$$\alpha.P \xrightarrow{\alpha} P$$

$$P \xrightarrow{ux} P', Q \xrightarrow{\bar{u}y} Q' \implies P \mid Q \xrightarrow{x=y} P' \mid Q'$$

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

$$P \xrightarrow{\phi} P', z\phi x, z \neq x \implies (x)P \xrightarrow{\phi \backslash z} P'x/z$$

$$P \xrightarrow{\alpha} P', x \notin names(\alpha) \implies (x)P \xrightarrow{\alpha} (x)P'$$

$$P \equiv P', Q \equiv Q', P \xrightarrow{\alpha} Q \implies P \xrightarrow{\alpha} Q$$

$$P \xrightarrow{(y)ax} P', z \in \{x, y\}, a \notin \{z, \bar{z}\} \implies (z)P \xrightarrow{(zy)ax} P'$$

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).

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

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.

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.

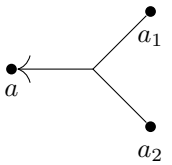
2.6 Solo Diagrams

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.

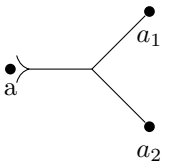
Definition 2.6.1. (Edge)
An edge is defined to be:

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

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



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



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

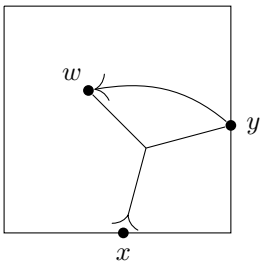
*For further details, Laneve and Victor (1999) discuss this implementation in Section 3

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.

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 $principals(M)$.

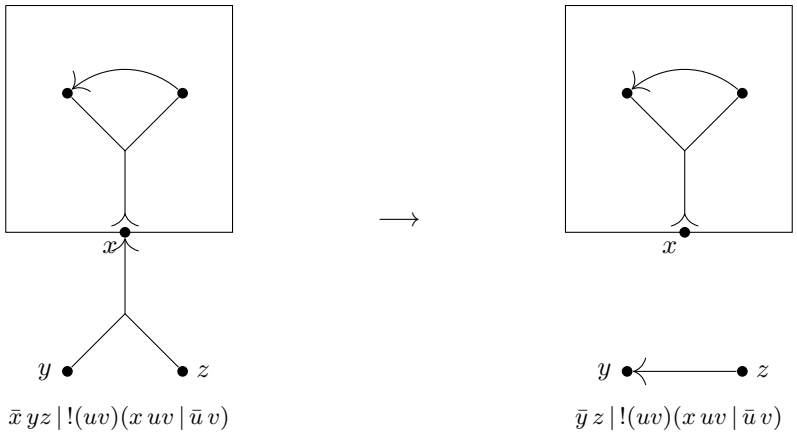
From here, we can convert Solo calculus to diagrams, where composition is intuitively just including two separate diagrams together and scope is simply any connected nodes labelled by ℓ. There are then four required reduction cases (edge-edge, edge-box, box-box and box internals) which can be deduced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

Definition 2.6.4. (Diagram Reduction)
Let $G, G_1, G_2 \dots$ be arbitrary graphs, M, M' arbitrary box multisets, $\alpha ::= \langle a, a_1 \dots a_k \rangle_i$, $\beta ::= \langle a, a'_1 \dots a'_k \rangle_o$, $\sigma ::= a_i \mapsto a'_i$, 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:

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

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

*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.



$\bar{x}\, yz \,|\, !(uv)(x\, uv \,|\, \bar{u}\, v)$ $\qquad$ $\bar{y}\, z \,|\, !(uv)(x\, uv \,|\, \bar{u}\, v)$

Example.

Remarks. 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 JavaScript

- *VivaGraph.js* for JavaScript
- *d3.js* for JavaScript

Remarks. Most of these libraries, especially those that allow interactivity with the output, are written for web-browsers and are subsequently written in JavaScript 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

4.1 Method and Results

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.

4.2 Calculus Implementation

4.2.1 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.

Lemma 4.2.1. 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 4.2.2. 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 4.2.3.
For any agent P, $\exists\, Q \equiv P$ such that Q is in normal form.

*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.

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

function NORMALISE(P)
  if Q a Scope **then**
    $(\tilde{bn})Q' ::= Q$
    $collisions ::= \tilde{bn} \cap names(P)$
    if $collisions \neq \emptyset$ **then**
      $\alpha : collisions \rightarrow fresh\ names$
      return NORMALISE(P, $\alpha(Q)$)
    else
      return $(\tilde{bn})$NORMALISE(P, Q')
    end if

  else if Q a Composition **then**
    $Q_1 \mid \ldots \mid Q_n ::= Q$
    return REDUCE(`Normalise`, P, $Q_1 \ldots Q_n$)

  else if Q a Replication **then**
    $Q' ::=$ NORMALISE(Q'') where $!Q'' ::= Q$
    if $replicators(Q') \neq \emptyset$ **then**
      $\bar{Q} ::=$ FLATTEN(Q)
      return NORMALISE(P, $\bar{Q}$)
    else
      $collisions ::= bn(P) \cap bn(Q')$
      $\alpha : collisions \rightarrow fresh\ names$
      return $P \mid !\,\alpha(Q')$
    end if

  else if Q a Solo **then**
    $(\tilde{bn})P' ::= P$
    $collisions ::= \tilde{bn} \cap names(Q)$
    if $collisions \neq \emptyset$ **then**
      $\alpha : collisions \rightarrow fresh\ names$
      return NORMALISE($\alpha(P)$, Q)
    else
      return $(\tilde{bn})(P' \mid Q)$
    end if
  end if
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 i ∈ inputs(P) do                    ▷ inputs(P) ≡ {x ∈ solos(P) | x an input}
3:         for o ∈ outputs(P) do               ▷ outputs(P) ≡ {x ∈ solos(P) | x an output}
4:             if i agrees with o then         ▷ 'agrees' ≡ matching subject and arity
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 ... i_n, o_1 ... o_n, bn)
2:     Graph g := {(i_j, o_j) | 1 ≤ j ≤ n}
3:     Map σ := id
4:     for each Graph ḡ ∈ partitions(g) do
5:         isect := nodes(ḡ) − bn
6:         if |isect| = 0 then
7:             fn := x ∈ nodes(ḡ)
8:         else if |isect| = 1 then
9:             fn := x ∈ isect
10:        else
11:            return none
12:        end if
13:        for each name ∈ nodes(ḡ) − {fn} do
14:            σ(name) := fn
15:        end for
16:    end for
17:    return σ
18: end function
```

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 i ∈ inputs(P) do
3:          ...
4:      end for
5:
6:      for i ∈ ⋃_{r∈reps(P)} inputs(r) do          ▷ reps(P) ≡ {x ∈ P | x a replicator}
7:          for o ∈ ⋃_{r∈reps(P)} outputs(r) do
8:              if i agrees with o then
9:                  σ := FUSE(i, o, bn(P) ∪ bn(r_i) ∪ bn(r_o))    ▷ r_x ≡ r ∈ reps(P) | x ∈ solos(r)
10:                 if σ ≠ none then
11:                     return P ∪ {expand(r_i), expand(r_o)}      ▷ expand(!R) ≡ R
12:                 end if
13:             end if
14:         end for
15:     end for
16:     return P
17: end function
```

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

4.2.2 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 4.3.2.

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*

```
1: function α-Equivalence(P, Q)
2:     if P ≡ Q then
3:         return id
4:     end if
5:     for comb_scope ∈ combinations(scope(P), scope(Q)) do
6:         α_{P,Q} : x ↦ y ∀ (x, y) ∈ comb_scope
7:         for comb_replicator ∈ combinations(replicators(P), replicators(Q)) do
8:             for each (r_P, r_Q) ∈ comb_replicator do
9:                 α_{r_P,r_Q} ::= α-Equivalence(r_P, r_Q)
10:            end for
11:            if none ∈ {α_{r_P,r_Q} | (r_P, r_Q) ∈ comb_replicator} then
12:                continue for
13:            else if ∃ α_1, α_2 ∈ {α_{r_P,r_Q}} s.t. α_1 disagrees with α_2 then
14:                continue for
15:            else
16:                α ::= α_{P,Q} ∪ ⋃_{α∈{α_{r_P,r_Q}}} α
17:            end if
18:            if α(P) ≡ α(Q) then
19:                return α
20:            end if
21:        end for
22:        if α(P) ≡ α(Q) then
23:            return α
24:        end if
25:    end for
26:    return none
27: end function
```

4.2.3 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 4.2.1.

4.3 Read-Eval-Print-Loop Implementation

4.3.1 Analysis

A simple REPL interface was constructed to interact with the code written in 4.2. 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 4.3.1. (Solo Matching)
Solos are built using the regex as follows:

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

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 4.3.2. (Replication)
Replicators are built using the regex as follows:

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

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

Definition 4.3.3. (Scope)
Scopes are built using the regex as follows:

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

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

Definition 4.3.4. (Composition)
Compositions are built using the recursive regex as follows:

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

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.

4.3.2 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. Finally, a test of an expression containing all of the above agents was reduced and checked for correctness.

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 10.1

4.3.3 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 5.3. Through this, the diagram presented by the server could be changed on the server-side.

5 Solo Diagrams

5.1 Methods and Results

The REST server then presents a computer-useable interface for reduction of calculus terms, as well as allowing input through the aforementioned REPL. The REST server alone serves no purpose, but combined with the diagram visualiser completes the program and allows for visualising inputted calculus terms as diagrams.

5.2 Diagram Implementation

pass

5.3 REST Server

6 Diagram Visualisation

6.1 Results

6.2 Visualiser

7 Conclusion

8 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.

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.

9 Agreement

9.1 Student Signature

9.2 Supervisor Signature

10 Appendix

10.1 Calculus Testing

Output of *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))
======================================================================
TestInterReplicatorFusion

 TestInterReplicatorFusion.TestOneBoundInner
()(p x y  !(x)(u x  u y)) -> ()(p x y  !(x)(u x  u y))

 TestInterReplicatorFusion.TestOneBoundOuter
(x)(p x y  !()(u x  u y)) -> ()(p y y  u y  u y  !()(u y  u y))

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

 TestInterReplicatorFusion.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))
```