A Graphical Representation of the Solos Calculus

CM30082 Dissertation

Adam Lassiter

May 2018

Abstract

The project seeks to provide an implementation of both the Solos calculus, for which there exists an encoding of the π-calculus within itself, and also of Solos 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. Furthermore, the design of the Solos 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 Solos 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 Solos calculus is found to be an interesting alternative to the more common π-calculus and is therefore chosen as the project focus.

Contents

1 Introduction

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. While when it was first designed, it was not expected to be relevant to the newly-emerging field of theoretical Computer Science but instead Discrete Mathematics, 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:

$$x[y ::= P] ::= \begin{cases} P & \text{if } x = y \\ x & \text{otherwise} \end{cases} \tag{1}$$

$$(\lambda x.M)[y ::= P] ::= \begin{cases} \lambda x.M & \text{if } x = y \\ \lambda x.(M[y ::= P]) & \text{if } x \neq y \text{ and } x \notin FV(P) \ ^{\dagger} \end{cases} \tag{2}$$

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:

$$y \notin FV(M) \implies \lambda x.M \equiv_\alpha \lambda y\,(M[x ::= y]) \tag{3}$$

$$M \equiv_\alpha M' \implies \begin{cases} M\,P \equiv_\alpha M'\,P \\ P\,M \equiv_\alpha P\,M' \\ \lambda x.M \equiv_\alpha \lambda x.M' \end{cases} \tag{4}$$

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:

$$(\lambda x.P)\,Q \to_\beta P[x ::= Q] \tag{5}$$

$$M \to_\beta M' \implies \begin{cases} P\,M \to_\beta PM' \\ M\,P \to_\beta M'P \\ \lambda x.M \to_\beta \lambda x.M' \end{cases} \tag{6}$$

*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:

$$M \rightarrow_\beta^* P \text{ and } N \rightarrow_\beta^* P \implies M \equiv_\beta N \tag{7}$$

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:

$$[x_1, \ldots, x_n] ::= \lambda c.\lambda n.(c\, x_1\, (\ldots (c\, x_n\, n) \ldots)) \tag{8}$$

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 $x_1 \ldots x_n$.

$$map ::= \lambda f.\lambda l.(\lambda c.(l\, (\lambda x.c\, (f\, x)))) \tag{9}$$

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.*

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

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

set of values, similar to that described above, and a *reduce* function that collects all values with matching keys. The model has found to be useful for modelling many real-world tasks for performance reasons, but concurrent calculus 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 (or just 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. Amongst 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:

$$ax_1 \dots x_n.P \xrightarrow{av_1 \dots v_n} P[v_1/x_1 \dots v_n/x_n] \tag{10}$$

$$\bar{a}v_1 \dots v_n.P \xrightarrow{\bar{a}v_1 \dots v_n} P \tag{11}$$

$$\tau.P \xrightarrow{\tau} P \tag{12}$$

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

$$P \xrightarrow{a} P' \implies \begin{cases} P + Q \xrightarrow{a} P' \\ P \mid Q \xrightarrow{a} P' \end{cases} \tag{13}$$

$$\left.\begin{array}{l} P \xrightarrow{a} P' \\ Q \xrightarrow{\bar{a}} Q' \end{array}\right\} \implies P \mid Q \xrightarrow{\tau} P' \mid Q' \tag{14}$$

*The reader is recommended to read '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 $dom(S) = \{a\}\, ran(S) = \{b\}$

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:

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

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:

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

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:

$$P \xrightarrow{ax} P' \implies \begin{cases} P\backslash b \xrightarrow{ax} P'\backslash b & \text{if } a \notin \{b, \bar{b}\} \\ P[S] \xrightarrow{S(a)x} P'[S] \end{cases} \tag{15}$$

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:

$$P[v_1/x_1 \dots v_n/x_n] \xrightarrow{ax} P' \implies b(v_1 \dots v_n) \xrightarrow{ax} P' \tag{16}$$

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:

$$\begin{array}{rcll} P & ::= & Q \mid R & \text{(concurrency)} \\ & & Q + P & \text{(choice)} \\ & & c(x).Q & \text{(input prefix)} \\ & & \bar{c}(x).Q & \text{(output prefix)} \\ & & \tau.P & \text{(silent prefix)} \\ & & \text{if } x = y \text{ then } P & \text{(match)} \\ & & vx.Q & \text{(restriction)}\ ^{*} \\ & & Q! & \text{(replication)}\ ^{\dagger} \\ & & \mathbf{0} & \text{(null process)} \end{array}$$

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 $a, b, c \ldots$ 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. $\ldots P$ 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.*

---

*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.

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:

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

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:

$$(x)\, \mathbf{0} \equiv \mathbf{0} \tag{17}$$

$$(x)\, (y)\, P \equiv (y)\, (x)\, P \tag{18}$$

$$(x)\, (P \mid Q) \equiv P \mid (x)Q \quad \text{where } z \notin fn(P) \tag{19}$$

$$(x)(P + Q) \equiv (x)P + (x)Q \tag{20}$$

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:

$$\alpha.P \xrightarrow{\alpha} P \tag{21}$$

$$P \xrightarrow{ux} P', Q \xrightarrow{\bar{u}y} Q' \implies P \mid Q \xrightarrow{x=y} P' \mid Q' \tag{22}$$

$$P \xrightarrow{\alpha} P' \implies \begin{cases} P \mid Q \xrightarrow{\alpha} P' \mid Q \\ P + Q \xrightarrow{\alpha} P' \end{cases} \tag{23}$$

$$P \xrightarrow{\phi} P', z\phi x, z \neq x \implies (x)P \xrightarrow{\phi \backslash z} P'x/z \tag{24}$$

$$P \xrightarrow{\alpha} P', x \notin names(\alpha) \implies (x)P \xrightarrow{\alpha} (x)P' \tag{25}$$

$$P \equiv P', Q \equiv Q', P \xrightarrow{\alpha} Q \implies P \xrightarrow{\alpha} Q \tag{26}$$

$$P \xrightarrow{(y)ax} P', z \in \{x, y\}, a \notin \{z, \bar{z}\} \implies (z)P \xrightarrow{(zy)ax} P' \tag{27}$$

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:

α	::=	$u\,\tilde{x}$	(input)
		$\bar{u}\,\tilde{x}$	(output) *
P	::=	0	(inaction)
		α	(solo)
		$Q \mid R$	(composition)
		$(x)\,Q$	(scope)
		$[x = y]\,Q$	(match)
		$!\,P$	(replication)

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.

Definition 2.5.2. (Match Operator)
The match operator $[x = y]\,P$ computes P if x and y are the same name, otherwise computes $\mathbf{0}$. These match operators are iterated over here by M, N, with sequences of match operators iterated over by $\tilde{M}, \tilde{N}$. Each name occurring in M is a *labelled node* of M.

Definition 2.5.3. (Structural Congruence)
The structural congruence relation $\equiv$ in the solo calculus is exactly that defined in Definition 2.4.2.

Definition 2.5.4. (Reduction)
Reduction semantics on solo expressions are defined as:

$$(x)(\bar{u}\,x \mid u\,y \mid P) \to P\{y/x\} \tag{28}$$

$$P \to P' \implies \begin{cases} P \mid Q \to P' \mid Q \\ (x)\,P \to (x)\,P' \\ P \equiv Q \text{ and } P' \equiv Q' \implies Q \to Q' \end{cases} \tag{29}$$

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.

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.

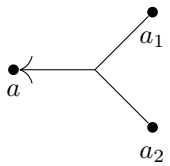
2.6 Solo Diagrams

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.

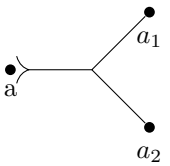
Definition 2.6.1. (Edge)
An edge is defined to be:

$$E ::= \langle a, a_1 \ldots a_k \rangle_t \quad \text{for } t \in \{i, o\} \tag{30}$$

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



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



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

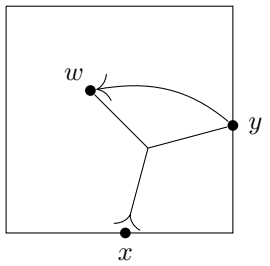
This is analogous to an input or output solo in the calculus, where a is u or $\bar{u}$ and $a_1 \ldots a_n$ is $\tilde{x}$ as written in Definition 2.5.1. Note that inputs and outputs must have matching arity — a 2-arity input cannot communicate with a 3-arity output for obvious reasons.

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

Definition 2.6.2. (Box)
A box is defined to be:

$$B ::= \langle G, S \rangle \quad \text{for } S \subset nodes(G)^* \tag{31}$$

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:

$$SD ::= (G, M, \ell) \tag{32}$$

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:

$$(G \cup \{\alpha, \beta\}, M, \ell) \to (G\sigma, M\sigma, \ell') \tag{33}$$

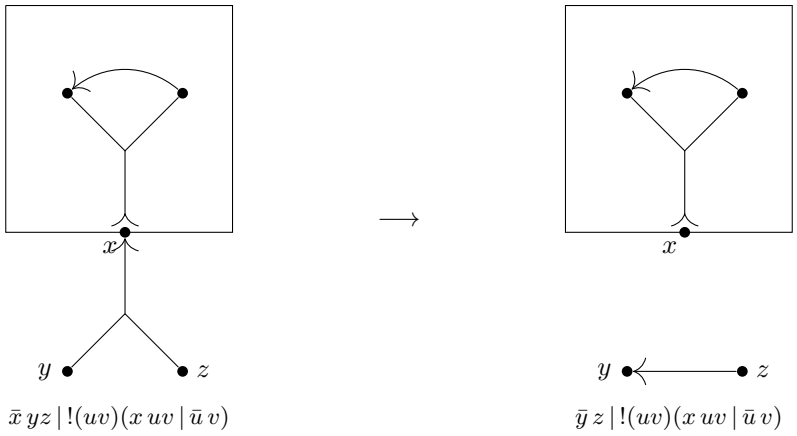
$$(G_1 \cup \{\alpha\}, M ::= \langle G_2 \cup \{\beta\}, S \rangle, \ell) \to ((G_1 \cup G_2\rho)\sigma, M\sigma, \ell') \tag{34}$$

$$(G, M ::= \{\langle \{\alpha\} \cup G_1, S_1 \rangle, \langle \{\beta\} \cup G_2, S_2 \rangle\} \cup M', \ell) \to ((G \cup G_1\rho \cup G_2\rho)\sigma, M\sigma, \ell') \tag{35}$$

$$(G, M ::= \langle \{\alpha, \beta\} \cup G_1, S \rangle \cup M', \ell) \to ((G \cup G_1\rho)\sigma, M\sigma, \ell') \tag{36}$$

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 complicated and 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 with the web-browser in mind 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.

4 Implementation

4.1 Solo Calculus

4.2 Solo Diagrams

4.3 Visualisation

5 Conclusion

6 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.

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.

*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.

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.

7 Agreement

7.1 Student Signature

7.2 Supervisor Signature