A Graphical Representation of the Solos Calculus

CM30082 Literature Review

Adam Lassiter

October 2017

Contents

1 Literature

1.1 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 1.1.1. (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 1.1.2. (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 1.1.3. (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 1.1.4. (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 1.1.5. (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.

1.2 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 1.3, 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 1.2.1. (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 1.1.1, 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 1.2.2. (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 1.2.3. (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.

Definition 1.2.4. (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 1.2.5. (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.

1.3 π-calculus

1.3.1 Background

Parrow (2001) Similar to the λ-calculus as described in 1.1, there exists the π-calculus for the study of concurrent computation.

1.3.2 Definitions

The π-calculus is constructed from the recursive definition of an expression P:

$$\begin{array}{lll} P & ::= & Q \mid R \quad \text{(concurrency)} \\ & & c(x).Q \quad \text{(input)} \\ & & \bar{c}(x).Q \quad \text{(output)} \\ & & vx.Q \quad \text{(name binding)} \ ^{*} \\ & & Q! \quad \text{(replication)} \ ^{\dagger} \\ & & 0 \quad \text{(null process)} \end{array}$$

1.3.3 Examples

1.3.4 Evaluation

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.

1.4 Fusion Calculus

1.4.1 Background

Miculan (2008)

1.4.2 Definitions

1.4.3 Examples

1.4.4 Evaluation

1.5 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 1.5.1. (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:

$$\begin{array}{lll} \alpha & ::= & u\,\tilde{x} \quad \text{(input)} \\ & & \bar{u}\,\tilde{x} \quad \text{(output)} \ ^{\ddagger} \\ \\ P & ::= & 0 \quad \text{(inaction)} \\ & & \alpha \quad \text{(solo)} \\ & & Q \mid R \quad \text{(composition)} \\ & & (x)\,Q \quad \text{(scope)} \\ & & [x = y]\,Q \quad \text{(match)} \\ & & !\,P \quad \text{(replication)} \end{array}$$

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).

*Name-binding in the π-calculus is similar to $\lambda x.P$ within the λ-calculus.

†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.

‡$\tilde{x}$ 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 1.5.2. (Match Operator)
The match operator $[x = y]\,P$ makes P behave as if x and y are the same name. 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 1.5.3. (Structural Congruence)
The structural congruence $\equiv$ is the least congruence relation satisfying α-equivalence, associativity, commutativity, 0 identity and the following scope laws:

$$(x)\,0 \equiv 0 \tag{17}$$
$$(x)\,(y)\,P \equiv (y)\,(x)\,P \tag{18}$$
$$[x = x]\,P \equiv P \tag{19}$$
$$(x)\,MP \equiv M\,(x)P \quad \text{if } x \notin \text{labelled nodes of M} \tag{20}$$
$$(x)(P \mid Q) \equiv P \mid (x)Q \quad \text{if } x \notin fn(P) \tag{21}$$

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

Definition 1.5.4. (Reduction)
Reduction semantics on solo expressions are defined as:

$$(x)(\bar{u}\,x \mid u\,y \mid P) \to P\{y/x\} \tag{22}$$
$$P \to P' \implies \begin{cases} P \mid Q \to P' \mid Q \\ (x)\,P \to (x)\,P' \\ P \equiv Q \text{ and } P' \equiv Q' \implies Q \to Q' \end{cases} \tag{23}$$

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. Hence there exists an encoding of the π-calculus also, complete with the same style of guarded input/output communication.

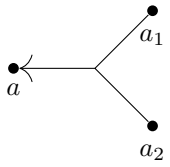
1.6 Solo Diagrams

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.

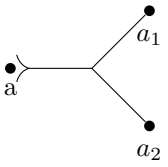
Definition 1.6.1. (Edge)
An edge is defined to be:

$$E ::= \langle a, a_1 \ldots a_k \rangle_t \quad \text{for } t \in \{i, o\} \tag{24}$$

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 $\langle a, a_1, a_2 \rangle_o$



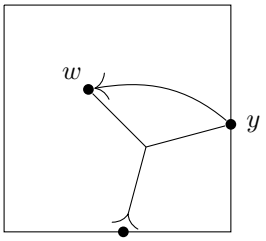
Input edge $\langle a, a_1, a_2 \rangle_i$

This is analogous to an input or output solo in the calculus, where a is u or $\bar{u}$ and $a_1 \dots a_n$ is $\tilde{x}$ as written in Definition 1.5.1. Note that inputs and outputs must have matching arity — a 2-arity input cannot communicate with a 3-arity output for obvious reasons.

Definition 1.6.2. (Box)
A box is defined to be:

$$B ::= \langle G, S \rangle \quad \text{for } S \subset nodes(G)^* \tag{25}$$

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 1.6.3. (Diagram)
A solo diagram is defined to be:

$$SD ::= (G, M, \ell) \tag{26}$$

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.

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.

2 Technology

2.1 Calculus Reduction

2.2 Diagram Visualisation

*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.

3 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.

Miculan, Marino. A Categorical Model of the Fusion Calculus. In *Electronic Notes in Theoretical Computer Science*, vol. 218(1):275–293 (2008). ISSN 1571-0661.

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, 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.