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), described by Robin Milner in 1980 in a book of the same name, 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 waiting on input/output from (usually) internal and (potentially) external sources. 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)
While use of syntax varies between sources, the core parts of CCS remain consistent across all of them, with some also including some extra syntax for clarity. The definition of CCS here is taken from Xu (2009), in particular the **Plain CHOCS** syntax. A process p is defined as:

p	::=	nil	(inactive process)
		x	(variable)
		$a?\,x.p$	(input)
		$a!\,p'.p$	(output)
		$\tau.p$	(silent)
		$p \mid p'$	(parallel composition)
		$(a)p$	(restriction)
		$p[a \to b]$	(relabelling) †
		$p + p'$	(choice) ‡
		$p!$	(replication)

When comparing to the λ-calculus from Definition 1.1.1, certain parallels can be seen. Notable additions are the action operators (input, output and silent) and the composition operators (parallel composition and choice).

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

Definition 1.2.2. (Basic Semantics)
The first set of rules, pertaining to just the actions, is simply:

$$\tau.p \xrightarrow{\tau} p \tag{10}$$

$$a?x.p \xrightarrow{a?x} p \tag{11}$$

$$a!p'.p \xrightarrow{a!_{\emptyset}p'} p \tag{12}$$

That is, a process of action λ and subprocess p written $\lambda.p$ is reduced to just p once the action λ is taken. Note also that the output operation includes a set, here it is the empty set $\emptyset$, which describes the bound names for the output process p'.

*The reader is recommended to read 'Communicating Sequential Processes' by Hoare (1978), which may be thought of as the first such concurrent process calculus

†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 will later be described as the Tea/Coffee Problem

Definition 1.2.3. (General Actions)
The next set of rules, describing a general action λ, is:

$$p \xrightarrow{\lambda} p' \implies \begin{cases} p + q \xrightarrow{\lambda} p' \\ p \,|\, q \xrightarrow{\lambda} p' \,|\, q \quad \text{if } bn(\lambda) \cap fn(q) = \emptyset \end{cases} \tag{13}$$

This describes how the composition operators reduce after actions are taken. The only catch here is the condition that the action λ cannot bind any of the free names in q.

Definition 1.2.4. (Silent Action)
The first special case of the input actions is the silent operation. These rules are simply:

$$p \xrightarrow{\tau} p' \implies \begin{cases} (c)p \xrightarrow{\tau} (c)p' \\ p[S] \xrightarrow{\tau} p'[S] \end{cases} \tag{14}$$

This can be thought of as similar to α-equivalence from the Corollary to Definition 1.1.2.

Definition 1.2.5. (Input Action)
The input action then has the special rules:

$$p \xrightarrow{a?x} p' \implies \begin{cases} (c)p \xrightarrow{a?x} (c)p' & \text{if } a \neq c \\ p[S] \xrightarrow{S(a)?x} p'[S] \end{cases} \tag{15}$$

That is, any restrictions or relabelling in p is carried through to p' after the action is taken as long as the action does not input on the restriction (first case). Similarly, the action is equivalent if, after renaming, the input variable a is renamed before the input action is taken.

Definition 1.2.6. (Output Action)
Then finally, the output operation has the rules:

$$p \xrightarrow{a!_B p''} p' \implies \begin{cases} p \xrightarrow{a!_{B'} p''} p' & \text{if } B \cap (fn(p') \cup fn(p'')) = B' \cap (fn(p') \cup fn(p'')) \\ (c)p \xrightarrow{a!_{B \cup \{c\}} p''} (c)p' & \text{if } a \neq c,\ c \in fn(p'') \text{ and } c \notin B \\ (c)p \xrightarrow{a!_B p''} (c)p' & \text{if } a \neq c \text{ and } c \notin fn(p'') \\ p[S] \xrightarrow{S(a)!_B p''} p'[S] & \text{if } B \cap (dom(S) \cup ran(S)) = \emptyset \end{cases} \tag{16}$$

The first of these states that the computation is equivalent for different B and B' as long as the free names of p' and p'' that are found in B are exactly those found in B'. The second and third describe carrying over restriction into p'' by including c in the bound names B as long as there isn't an obvious name collision, with each describing what to do depending on whether c is free in p''. Finally, the fourth states much the same, but as to renaming rather than restriction, in combination with Equation 15 — computation is equivalent under renaming if a is renamed prior to performing the action and there are no name collisions between B and S.

Definition 1.2.7. (Concurrent Inputs and Outputs)
The final semantic rule describes the communication of two concurrent processes:

$$\left.\begin{array}{l} p \xrightarrow{a!_B p''} p' \\ q \xrightarrow{a?x} q' \end{array}\right\} \implies p \,|\, q \xrightarrow{\tau} (B)(p' \,|\, q'\{p''/x\}) \quad \text{if } fn(q) \cap B = \emptyset \tag{17}$$

If one process $q \xrightarrow{\lambda} q'$ makes an input action $\lambda ::= a?x$ and another process $p \xrightarrow{\mu} p'$ makes an output action $\mu ::= a!_B p''$ then the parallel computation $p \,|\, q$ is equivalent to:

- Restrict in B
- Rename p" to x
- Compute $p'|q'$

This is assuming, of course, there are no free names of q that appear in B.

1.3 π-calculus

1.3.1 Background

Parrow (2001) Similar to the λ-calculus as described in 1.1.1, 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:

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

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 Solos Calculus

1.5.1 Background

Ehrhard and Laurent (2010)

[†]Name-binding in the π-calculus is similar to $\lambda x.P$ within the λ-calculus.
[†]Replication is defined in theory as $P! ::= P \mid P!$.

1.5.2 Definitions

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

$$
\begin{array}{lcll}
\alpha & ::= & u\tilde{x} & \text{(input)} \\
 & & \bar{u}\tilde{x} & \text{(output)} \\
 & & & \\
P & ::= & 0 & \text{(inaction)} \\
 & & \alpha & \text{(solo)} \\
 & & Q|R & \text{(composition)} \\
 & & (x)Q & \text{(scope)} \\
 & & [x = y]Q & \text{(match)}
\end{array}
$$

1.5.3 Examples

1.5.4 Solos Diagrams

This 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.

2 Technology

2.1 Calculus Reduction

2.2 Diagram Visualisation

Graf et al. (2008)

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 18605974.

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 00010782.

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.

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.