Self-replicating Expressions in the Lambda Calculus

James Larkin **Phil Stocks**

School of Information Technology
Bond University,
Gold Coast, Queensland 4229
Australia
Email: {jalarkin,pstocks}@staff.bond.edu.au

Abstract

The study of self-replicating structures in Computer Science has been taking place for more than half a century, motivated by the desire to understand the fundamental principles and algorithms involved in self-replication. The bulk of the literature explores self-replicating forms in Cellular Automata. Though trivially self-replicating programs have been written for dozens of languages, very little work exists that explores self-replicating forms in programming languages.

This paper reports initial investigations into self-replicating expressions in the Lambda Calculus, the basis for functional programming languages. Mimicking results from the work on Cellular Automata, self-replicating Lambda Calculus expressions that also allow the application of an arbitrary program to arbitrary data are presented. Standard normal order reduction, however, will not reduce the sub-expression representing the program application. Two approaches of dealing with this, hybrid reduction and parallel reduction, are discussed, and have been implemented in an interpreter.

Keywords: Self-replicating programs, Lambda Calculus, Functional Programming, Cellular Automata

1 Introduction

Two main goals of studying self-replicating programs are to improve understanding of the basic principles and algorithms of self-reproduction and to develop machines displaying or mimicking such biological properties as self-reproduction, self-repair, growth and evolution.

The history of self-replicating programs begins with John von Neumann who contrived a cellular automaton that took some input, and produced as output that input (von Neumann 1966). Therefore, if the automaton itself was given as input, the automaton would reproduce itself as output.

Cellular Automata are dynamic systems in which space and time are discrete. They consist of an array of cells, each of which can be in one of a finite number of possible states. The state of each cell is determined by the previous state of the surrounding cells, usually specified in a rule table. Thus, each cell has an underlying finite state automaton. Mitchell (1996) is a good overview of the field.

Von Neumann's automaton is an example of what is known as *trivial* self-replication because the structure to reproduce is encoded directly within the program or the input. This kind of trivial self-replication is easily implemented in any programming language. The *Quine Page* (Thompson) has a comprehensive list of such programs for dozens of languages.

Von Neumann's automaton has the capabilities of both universal construction and computation. Universal computation is the ability to execute any computational task. Universal construction is the ability to construct any kind of configuration in the cellular space. Langton disregarded the capabilities of universal construction and computation and created an automaton capable of non-trivial self-replication (Langton 1984, Langton 1986). The automaton is not passed as input to itself, nor directly encoded within itself.

Tempesti expanded on the capabilities of Langton's automaton, by creating an automaton that first replicates itself and then executes a directly encoded static program (Tempesti 1995). Tempesti's automaton achieves self-replication in almost exactly the same way as Langton's automaton, and does not have the capabilities of universal construction and computation. Tempesti's automaton is not able to execute an arbitrary given program.

Perrier, Sipper, and Zahnd's goal was to create an automaton capable of universal construction and computation (Perrier, Sipper & Zahnd 1996). They built on the concepts found in Langton's automaton and added a program and data stream into the automaton. Their automaton first reproduces itself along with the given program and data, and then executes the included program on the given data.

Outside the field of Cellular Automata, there is very little work on self-replicating programs beyond the most simple form of trivial replication. One very interesting work by McKay and Essam (2001) explores self-replicating structures in (functional) programming languages. The goal of their work was to determine whether a program can evolve to be self-replicating, from some initial random population of programs. They used genetic algorithms to search for self-replicating structures in a functional language of their own devising. Their aim was to gain some insight into the algorithmic aspects of the necessary characteristics of the evolution of life.

Due to the complexity of Cellular Automata, it is hard to explore features of self-replicating structures implemented in cellular automata. The overall goal of this work is to explore self-replication in the structurally simpler language of the Lambda Calculus. Initial investigations into a self-replicating expression in the Lambda Calculus that also applies an arbitrary program to arbitrary data revealed the problem that normal-order reduction (the standard for the Lambda Calculus) will not execute the program on the data

in the self-replicating expression. This paper presents the self-replicating Lambda Calculus expression supporting program application and several interesting variations of it, and then shows how the question of executing the program expression can be addressed using strict-order reduction, a hybrid of strict and normal order reduction, and parallel reduction.

First, a brief overview of the relevant features of the Lambda Calculus is presented.

2 Overview of the Lambda Calculus

Church's work on the Lambda Calculus was motivated by the desire to create a calculus of the behaviour of functions. The Lambda Calculus captures functions in their fullest generality. Functions are also values. Functions can therefore be applied to functions. A function (or lambda abstraction) is introduced by the symbol λ followed by the argument and the body of the function. For example, the expression $(\lambda a.\ +\ a\ 1)$ represents the successor function (using prefix notation) that takes an argument a and adds 1 to it. The application of this function to an actual argument, say 0, is written $(\lambda a.\ +\ a\ 1)\ 0$. Functions of more than one argument are Curried into expressions using successive applications of functions of one argument. For example, a function that takes two numbers and adds them together would be written $(\lambda x.(\lambda y.\ +\ x\ y))$. Parameter names are bound in the scope of the lambda abstraction. This section only provides an overview of aspects of the Lambda Calculus relevant to this paper. For more information on the Lambda Calculus see, for example, Barendregt (1984) or Peyton-Jones (1987).

A functional program is an expression. A functional program is executed by evaluating the expression that the program represents. Evaluation of an expression proceeds by repeatedly selecting a reducible sub-expression and reducing it until there are no more reducible sub-expressions. There are three main techniques used to reduce expressions: alpha-conversion, eta-conversion, and beta-reduction.

Alpha-conversion defines the naming equivalence between two lambda abstractions. For example, the two abstractions $(\lambda x.x)$ and $(\lambda y.y)$ are equivalent. Eta-conversion is a rule expressing the semantic equivalence of two lambda abstractions. For example, the two expressions $(\lambda x.+1\ x)$ and $(+\ 1)$ behave in exactly the same way when applied to some argument.

Beta-reduction is the main reduction technique. Beta-reduction occurs when there is a function application. This includes all the built-in operators (such as arithmetic operators) and the reduction of lambda abstractions. Beta-reduction is the replacement of the formal parameters in the body of the lambda abstraction by copies of the arguments to the abstraction. For example, $(\lambda a.\ +\ a\ 1)\ 0$ beta-reduces to $(+\ 0\ 1)$, which then beta-reduces to 1.

A Lambda Calculus expression that has no reducible expressions is said to be in *normal form*. When an expression contains more than one reducible expression, the question arises of which one to reduce first. There are two main reduction orders: normal order reduction and applicative order reduction (also called strict order reduction). Normal order reduction specifies that the leftmost, outermost reducible expression is reduced first. An outermost reducible expression is one that is not contained within any others. Applicative order reduction specifies that the leftmost, innermost reducible expression is reduced first. An innermost reducible expression is one that contains no others. The significance of applicative order reduction is that the argument to a function will be reduced (evaluated) before being passed into the function parameter. Using normal order reduction, the argument expression is passed in unevaluated.

Below is a simple expression containing two reducible sub-expressions, which are underlined.

$$\underline{(\lambda a.\ +\ a\ 1)}\ \underline{(+\ 2\ 3)}$$

Reducing this expression using normal order reduction yields this reduction sequence

$$(\lambda a.\ +\ a\ 1)\ (+\ 2\ 3) \rightarrow (+(+\ 2\ 3)\ 1) \rightarrow (+\ 5\ 1) \rightarrow 6$$

Reducing this expression using applicative order reduction yields this reduction sequence

$$(\lambda a.\ +\ a\ 1)\ (+\ 2\ 3) \rightarrow (\lambda a.\ +\ a\ 1)\ 5 \rightarrow (+\ 5\ 1) \rightarrow 6$$

Church-Rosser Theorems I and II describe the ramifications of reduction order (see, for example, Peyton-Jones (1987)). Church-Rosser Theorem I states that normal order reduction and applicative order reduction of the same expression cannot produce two different normal forms. Church-Rosser Theorem II states that if an expression has a normal form, then normal order reduction is guaranteed to find it, but applicative order reduction is not necessarily guaranteed to find it. The latter can happen in the case of an argument requiring infinite reduction that in actuality is not needed by the expression. A very simple example of this is the expression $(\lambda x.0)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$. Reducing this expression using normal order reduction yields this reduction sequence

$$(\lambda x.0)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \rightarrow 0$$

Reducing this expression using applicative order reduction yields this reduction sequence

$$(\lambda x.0)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \rightarrow (\lambda x.0)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \rightarrow \ldots$$

because $(\lambda x.x\ x)\ (\lambda x.x\ x)$ beta-reduces to itself.

3 Self-replicating Expressions in the Lambda Calculus

The most well-known self-replicating Lambda Calculus expression is $(\lambda x.x\ x)(\lambda x.x\ x)$. This expression beta-reduces to itself and therefore no normal form exists. This expression is conceptually similar to that of von Neumann's cellular automaton as both take the program as an argument and reproduce, as output, the input. As seen earlier, Perrier et al. and Tempesti added the capability of executing a program after self-replication had been achieved. This section discusses how to achieve this same goal in the Lambda Calculus.

3.1 A Self-replicating Pattern with Program Application

In order to apply a program to given data, there needs to be an expression representing the application of a program expression to a data expression. Let the program to be executed be π, and the data that the program will apply itself to be δ. In Perrier et al. and Tempesti, self-replication is achieved before the program is executed. The program and data are replicated to the new structure during this self-replicating stage. In the Lambda Calculus expression, the program and data will need to be arguments into the original self-replicating expression, so that both the

program and data replicate along with the original expression. Here is such an expression

$(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ \pi\ \delta$

Tracing the reduction of this expressions yields

$(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ \pi\ \delta$
$\rightarrow (\lambda p.(\lambda d.(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ p\ d))\ \pi\ \delta$
$\rightarrow (\lambda d.(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ \pi\ d)\ \delta$
$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d)))\ \pi\ \delta$

and it can be seen that the expression replicates itself. However, the program has not been applied to the data. In the Lambda Calculus, the application of program p to data d is represented by $(p\ d)$. Here is a modification of the previous expression that incorporates a program application expression

$(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d\ (p\ d))))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d\ (p\ d))))\ \pi\ \delta$

For textual brevity, let Ξ_0 represent the sub-expression $(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d\ (p\ d))))$ in the reduction below. Tracing the reduction of this expression yields

$\Xi_0\ \Xi_0\ \pi\ \delta$
$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ d\ (p\ d))))\ \Xi_0\ \pi\ \delta \quad [expanding\ \Xi_0]$
$\rightarrow (\lambda p.(\lambda d.\Xi_0\ \Xi_0\ p\ d\ (p\ d)))\ \pi\ \delta$
$\rightarrow (\lambda d.\Xi_0\ \Xi_0\ \pi\ d\ (\pi\ d))\ \delta$
$\rightarrow \Xi_0\ \Xi_0\ \pi\ \delta\ (\pi\ \delta)$

This expression has not replicated itself exactly, but it does contain a sub-expression representing the application of the input program π to δ. The immediate problem with this expression is that it is not well formed as the expression Ξ_0 takes three arguments but is being applied to four. This can be addressed by adding an extra lambda abstraction to the original expression. Let Ξ_1 represent $(\lambda x.(\lambda p.(\lambda d.(\lambda a.x\ x\ p\ d\ (p\ d)))))$. Using this as the stem for the self-replicating expression and adding a dummy initial argument α, yields

$\Xi_1\ \Xi_1\ \pi\ \delta\ \alpha$

Tracing the reduction of this expression yields

$\Xi_1\ \Xi_1\ \pi\ \delta\ \alpha$
$\rightarrow (\lambda x.(\lambda p.(\lambda d.(\lambda a.x\ x\ p\ d\ (p\ d)))))\ \Xi_1\ \pi\ \delta\ \alpha \quad [expanding\ \Xi_1]$
$\rightarrow (\lambda p.(\lambda d.(\lambda a.\Xi_1\ \Xi_1\ p\ d\ (p\ d))))\ \pi\ \delta\ \alpha$
$\rightarrow (\lambda d.(\lambda a.\Xi_1\ \Xi_1\ \pi\ d\ (\pi\ d)))\ \delta\ \alpha$
$\rightarrow (\lambda a.\Xi_1\ \Xi_1\ \pi\ \delta\ (\pi\ \delta))\ \alpha$
$\rightarrow \Xi_1\ \Xi_1\ \pi\ \delta\ (\pi\ \delta)$
...
$\rightarrow \Xi_1\ \Xi_1\ \pi\ \delta\ (\pi\ \delta)$
...

This expression is self-replicating and contains a sub-expression representing the application of a given program to given data. On each self-replication, this application expression is discarded and regenerated. Note, that using normal order reduction, the application of the program to the data is never reduced, so the program is never actually executed.

A more interesting expression is one which does not discard the application of program to data sub-expression, but rather uses this sub-expression in place of the data expression for the next self-replication cycle. This expression, which no longer needs the extra argument, is

$(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ (p\ d))))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ (p\ d))))\ \pi\ \delta$

Let Ξ_{fix} represent $(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ (p\ d))))$. To see the siginificance of using Ξ_{fix} as the stem in the self-replicating expression, consider the trace of its reduction:

$\Xi_{fix}\ \Xi_{fix}\ \pi\ \delta$
$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ (p\ d))))\ \Xi_{fix}\ \pi\ \delta \quad [expanding\ \Xi_{fix}]$
$\rightarrow (\lambda p.(\lambda d.\Xi_{fix}\ \Xi_{fix}\ p\ (p\ d)))\ \pi\ \delta$
$\rightarrow (\lambda d.\Xi_{fix}\ \Xi_{fix}\ \pi\ (\pi\ d))\ \delta$
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ \pi\ (\pi\ \delta)$

Again, exact self-replication is not achieved, but it incorporates the sub-expression of the program applied to the data. Also, as before, the application of the program to the data is never reduced in the normal order reduction of this expression. Nevertheless, the continued reduction of this expression produces an interesting pattern:

$\Xi_{fix}\ \Xi_{fix}\ \pi\ \delta$
...
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ \pi\ (\pi\ \delta)$
...
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ \pi\ (\pi\ (\pi\ \delta))$
...
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ \pi\ (\pi\ (\pi\ (\pi\ \delta)))$
...

Thus, this expression replicates its basic structure and builds the application of the program to the data to a fixed point.[1]

Naturally, this expression has some structural and conceptual similarity with the fixed-point combinator Y: $\lambda h.(\lambda x.h\ (x\ x))(\lambda x.h\ (x\ x))$. The replicating pattern in the Y combinator is to allow recursive functions to be simulated by fixed-point calculation. Self-replication of the original expression is not the goal. The normal-order reduction of the Y combinator applied to a suitable program and argument will calculate the fixed-point application of the program to the argument. By contrast, the self-replicating expression has the primary goal of self-replication, which leads to the problem described above that the application of the program to the data is never reduced.

Two approaches for forcing the reduction of the application of the program to the data are discussed in Section 3.3. First, though, the next section looks at some variations on the self-replicating expressions discussed above.

3.2 Variations on Self-replicating Expressions

The fixed point self-replicating expression builds the application of the program to the data to a fixed point by substituting the program application for the data expression in the replication stage. An alternative to this is to substitute the program application for the program expression in the replication stage. This

[1]Note, the expression using Ξ_{fix} subsumes the expression using Ξ_1. The Ξ_1 expression repeatedly generates the application $(\pi\ \delta)$. A bit of trickery using the Ξ_{fix} expression and the identity function, I, can simulate that behaviour: $\Xi_{fix}\ \Xi_{fix}\ I\ (\pi\ \delta) \rightarrow \Xi_{fix}\ \Xi_{fix}\ I\ (I\ (\pi\ \delta)) \rightarrow$
$\Xi_{fix}\ \Xi_{fix}\ I\ (I\ (I\ (\pi\ \delta))) \rightarrow \ldots$

expression is

$$(\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ \pi\ \delta$$

Tracing the reduction of this expression yields

$$(\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ \pi\ \delta$$
$$\ldots$$
$$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ (\pi\ \delta)\ \delta$$
$$\ldots$$
$$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ (p\ d)\ d)))\ ((\pi\ \delta)\ \delta)\ \delta$$
$$\ldots$$

The original program expression is lost and replaced with the application of the program to the data. If this application (when reduced) yields a value expression, the resulting expression is not well formed. However, if the application of the program to the data yields a function (program), this new function is now being applied to the data, which can potentially yield very interesting results.

Another variant on the original expression is to introduce a list structure for storing the results of program applications. Typically, $CONS$ is used to represent the function that adds something to the head of a list, and NIL is used to represent the empty list. See, for example, Peyton-Jones (1987) for the full definition of these functions in the Lambda Calculus. Let Ξ_{list} represent $(\lambda x.(\lambda p.(\lambda d.(\lambda l.x\ x\ p\ d\ (CONS\ (p\ d)\ l)))))$. The expression

$$\Xi_{list}\ \Xi_{list}\ \pi\ \delta\ NIL$$

builds the application of the program to the data, but rather than discarding it, or building it to a fixed point, puts the application onto the front of a list taken as an additional argument. Tracing its reduction yields

$$\Xi_{list}\ \Xi_{list}\ \pi\ \delta\ NIL$$
$$\ldots$$
$$\rightarrow \Xi_{list}\ \Xi_{list}\ \pi\ \delta\ (CONS\ (\pi\ \delta)\ NIL)$$
$$\ldots$$
$$\rightarrow \Xi_{list}\ \Xi_{list}\ \pi\ \delta\ (CONS\ (\pi\ \delta)\ (CONS\ (\pi\ \delta)\ NIL))$$
$$\ldots$$

The list being constructed is $((\pi\ \delta)\ (\pi\ \delta)\ldots(\pi\ \delta))$.

A small variation to Ξ_{list} incorporating the idea of Ξ_{fix} will enable the construction of the list of successive approximations to the fixed point application of the program to the data. Let Ξ_{fix_l} represent $(\lambda x.(\lambda p.(\lambda d.(\lambda l.x\ x\ p\ (p\ d)\ (CONS\ (p\ d)\ l)))))$. Tracing the reduction of the expression

$$\Xi_{fix_l}\ \Xi_{fix_l}\ \pi\ \delta\ NIL$$

yields

$$\Xi_{fix_l}\ \Xi_{fix_l}\ \pi\ \delta\ NIL$$
$$\ldots$$
$$\rightarrow \Xi_{fix_l}\ \Xi_{fix_l}\ \pi\ (\pi\ \delta)\ (CONS\ (\pi\ \delta)\ NIL)$$
$$\ldots$$
$$\rightarrow \Xi_{fix_l}\ \Xi_{fix_l}\ \pi\ (\pi\ (\pi\ \delta))\ (CONS\ (\pi\ (\pi\ \delta))\ (CONS\ (\pi\ \delta)\ NIL))$$
$$\ldots$$

The list being constructed is $(\ldots(\pi\ (\pi\ (\pi\ \delta)))\ (\pi\ (\pi\ \delta))\ (\pi\ \delta))$.

The variant expressions presented thus far have assumed no structure of the data expression. If the data expression is a list, more interesting variants are possible.[2] As an example of the potential of this, the expression

$$\Xi_{map}\ \Xi_{map}\ \pi\ \delta\ NIL$$

where Ξ_{map} represents

$$(\lambda x.(\lambda p.(\lambda d.(\lambda l.x\ x\ p\ (TAIL\ d)\ (CONS\ (p\ (HEAD\ d))\ l)))))$$

and δ is a list, simulates the standard *map* function, where a function is applied to each element of a list. This expression uses $HEAD$ and $TAIL$ to represent the standard list functions for accessing the first element of a list and the remainder of a list, respectively. Again, see for example Peyton-Jones (1987) for their definitions.

3.3 Reduction of Program Application in Self-replicating Expressions

The preceding sections introduce several variations on a self-replicating Lambda Calculus expression that also build the application of an arbitrary program to given data, but, using normal order reduction, the application of the program expression to the data expression is never reduced. This section discusses using applicative order reduction and parallel reduction to address this issue.

As a concrete example for use in this section, consider the successor function $(\lambda a.+\ a\ 1)$. The normal order reduction of the fixed point self-replicating expression using this function and the initial argument 0 is

$$\Xi_{fix}\ \Xi_{fix}\ (\lambda a.+\ a\ 1)\ 0$$
$$\ldots$$
$$\rightarrow \Xi_{fix}\ \Xi_{fix}\ (\lambda a.+\ a\ 1)\ ((\lambda a.+\ a\ 1)\ 0)$$
$$\ldots$$
$$\rightarrow \Xi_{fix}\ \Xi_{fix}\ (\lambda a.+\ a\ 1)\ ((\lambda a.+\ a\ 1)\ ((\lambda a.+\ a\ 1)\ 0))$$
$$\ldots$$

While the expression $((\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ 0))$ does indeed represent the value 2, this concrete value is never calculated through reduction.

3.3.1 Hybrid Normal and Applicative Order Reduction

One approach to achieving the reduction of the program application in the self-replicating expressions is simply to use applicative order reduction. Under applicative order reduction, the argument to a lambda abstraction will be reduced before the beta-reduction of the lambda abstraction. Figure 1 traces the applicative order reduction of the fixed point self-replicating expression applied to the successor function.

The expression replicates itself and reduces the application of an arbitrary program to the given input. Nevertheless, applicative order reduction generally is not preferred because it is not always guaranteed to reduce an expression to its normal form if the normal form exists. A particular example is a program that works on an infinite (or computationally prohibitive) list, but only needs the elements of the list one at a time to produce meaningful results. Normal order reduction allows such a scheme, but applicative order reduction will require the entire list to be generated before the program does any work.

A hybrid reduction scheme using both normal and applicative order reduction can, perhaps, provide the

[2] For a program on multiple arguments, having it work on a list of those arguments instead means the stem of the self-replicating expression won't have to expand to accommodate the extra arguments.

$\Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ 0$
...
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ 0)$
$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ (p\ d))))\ \Xi_{fix}\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ 0)$ [*expanding* Ξ_{fix}]
$\rightarrow (\lambda p.(\lambda d.\Xi_{fix}\ \Xi_{fix}\ p\ (p\ d)))\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ 0)$
$\rightarrow (\lambda d.\Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ d))\ ((\lambda a.+a\ 1)\ 0)$
$\rightarrow (\lambda d.\Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ d))\ 1$ [*argument* $((\lambda a.+a\ 1)\ 0) \rightarrow 1$ *before being passed to* d]
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ 1)$
...
$\rightarrow \Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ ((\lambda a.+a\ 1)\ 2)$
...

Figure 1: Applicative order reduction of $\Xi_{fix}\ \Xi_{fix}\ (\lambda a.+a\ 1)\ 0$

best of both worlds. The idea of reducing the arguments to a function before passing them is appealing because it solves the problem of the program application not being reduced in the self-replicating expressions. In the reduction of the fixed point self-replicating expression, the program application (π δ) becomes an argument to the main expression after self-replication. Using applicative order reduction on this one argument would mean that this sub-expression would be reduced before being passed into the main expression. This expression would be reduced using normal order evaluation as would the rest of the expression. The self-replicating reduction can proceed, as can the application of the program to the data. The data itself can potentially be an infinitely reducing structure because it does not need to be reduced using applicative order evaluation. Only the application of the program to the data requires that.

This hybrid reduction requires extending the Lambda Calculus with a special symbol representing applicative order evaluation. This *strictness symbol* is used to wrap a particular expression. If an expression is enclosed by the strictness symbol, then that expression is reduced before being passed into any lambda abstraction. Otherwise, normal order reduction is used. The strictness symbol thus provides the advantage of applicative order reduction when required, with normal order reduction as the default.[3] Note also that if the strictness symbol is applied to all arguments then fully applicative order reduction is implemented. So, the strictness symbol can be used to mimic applicative order reduction.

Hybrid reduction can solve the program application problem in the self-replicating expressions by wrapping the program application expression with the strictness symbol. Using \$ as the strictness symbol, the new fixed-point expression is

$$(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ \$(p\ d))))\ (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ \$(p\ d))))\ \pi\ \delta$$

Figure 2 traces the reduction of this expression using the successor function as the program, where $\Xi_{strictfix}$ represents $(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ \$(p\ d))))$. Note also that the successor function is modified to wrap its body with a strictness symbol in order to fully reduce the addition after the argument is passed in to a.

While hybrid reduction addresses the issue of reducing the program application, it requires extending the syntax and semantics of the Lambda Calculus.

[3]Contrast this with an applicative-order language such as Scheme with predefined functions `delay` and `force` to prohibit the evaluation of an argument until it is required, thus simulating normal-order evaluation.

3.3.2 Parallel Reduction

One of the most attractive features of functional programming languages is that they are not inherently sequential. At any time in the reduction of an expression, there can be a number of reducible expressions. In principle, all reducible expressions could be reduced simultaneously. A good discussion of parallel reduction can be found in Peyton-Jones (1987). The relevant issues are summarised below.

There are a number of possibilities as to which reducible expressions should be reduced simultaneously. One approach is to use *conservative parallelism*. This type of parallelism only reduces a sub-expression if it is known that the sub-expression will need to be reduced in order for the whole expression to reach its normal form. Conservative parallelism doesn't introduce much parellelsim since it is difficult to determine whether or not an arbitrary sub-expression will need to be reduced. A more useful method is *speculative parallelism*, which in its most liberal sense says that every reducible expression should be reduced even though it may not be required in the final normal form. Speculative parallelism substantially increases the number of expressions being reduced in parallel, but may also introduce much work that does not in the end need to be done.

Two problems faced when using speculative parallelism are expressions that don't have a normal form, and expressions that have a normal form but take a substantial amount of time or reduction steps to reach this form. The simplest solution to both of these problems is to reduce expressions in single steps, i.e., to reduce all the reducible expressions in an expression by one beta-reduction.

Figure 3 traces the reduction of the fixed point self-replicating expression using parallel reduction where all reducible expressions are reduced by one beta-reduction and with the successor function as the program. The reducible sub-expressions are underlined.

Using parallel graph reduction, the application of the program to the data is reduced. However the expression doesn't replicate its original form. Interestingly, the original expression reduces to a self-replicating form after three beta-reductions (two steps in the above parallel reduction). This means that the expression has essentially evolved into a self-replicating expression.

4 Implementation

The two reduction approaches discussed in Section 3.3 have been implemented in a simple Lambda Calculus interpreter written in the Scheme programming language. The interpreter also supports normal order

$\Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ 0$

$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ \$(p\ d))))\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ 0$ [*expanding* $\Xi_{strictfix}$]

$\rightarrow (\lambda p.(\lambda d.\Xi_{strictfix}\ \Xi_{strictfix}\ p\ \$(p\ d)))\ (\lambda a.\$(+\ a\ 1))\ 0$

$\rightarrow (\lambda d.\Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ d))\ 0$

$\rightarrow \Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ 0)$

$\rightarrow (\lambda x.(\lambda p.(\lambda d.x\ x\ p\ \$(p\ d))))\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ 0)$ [*expanding* $\Xi_{strictfix}$]

$\rightarrow (\lambda p.(\lambda d.\Xi_{strictfix}\ \Xi_{strictfix}\ p\ \$(p\ d)))\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ 0)$

$\rightarrow (\lambda d.\Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ d))\ \$((\lambda a.\$(+\ a\ 1))\ 0)$

$\rightarrow (\lambda d.\Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ d))\ 1$

$\rightarrow \Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ 1)$

...

$\rightarrow \Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+\ a\ 1))\ \$((\lambda a.\$(+\ a\ 1))\ 2)$

...

Figure 2: Hybrid reduction of $\Xi_{strictfix}\ \Xi_{strictfix}\ (\lambda a.\$(+a\ 1))\ 0$

$\Xi_{fix}\ \Xi_{fix}\ (\lambda a.\ (+\ a\ 1))\ 0$

$\underline{(\lambda x.(\lambda p.(\lambda d.x\ x\ p\ (p\ d))))\ \Xi_{fix}}\ (\lambda a.\ (+\ a\ 1))\ 0$

$\rightarrow \underline{(\lambda p.(\lambda d.\Xi_{fix}\ \Xi_{fix}\ p\ (p\ d)))\ (\lambda a.\ (+\ a\ 1))}\ 0$

$\rightarrow \underline{(\lambda d.\underline{(\lambda p.(\lambda d.\Xi_{fix}\ \Xi_{fix}\ p\ (p\ d)))\ (\lambda a.+\ a\ 1)}\ \underline{((\lambda a.+\ a\ 1)\ d)})}\ 0$

$\rightarrow \underline{(\lambda d.\underline{(\lambda p.(\lambda d.\Xi_{fix}\ \Xi_{fix}\ p\ (p\ d)))\ (\lambda a.+\ a\ 1)}\ \underline{((\lambda a.+\ a\ 1)\ d)})}\ 1$

$\rightarrow \underline{(\lambda d.\underline{(\lambda p.(\lambda d.\Xi_{fix}\ \Xi_{fix}\ p\ (p\ d)))\ (\lambda a.+\ a\ 1)}\ \underline{((\lambda a.+\ a\ 1)\ d)})}\ 2$

...

Figure 3: Parallel reduction of $\Xi_{fix}\ \Xi_{fix}\ (\lambda a.\ (+\ a\ 1))\ 0$

reduction. Further details on the implementation can be found in Larkin (2003).

5 Conclusions

This paper reports preliminary investigations into self-replicating Lambda Calculus expressions. The goal is to explore self-replicating patterns in (functional) programming languages. In contrast to McKay and Essam, who used genetic algorithms to evolve self-replicating programs in a functional language, this work follows the path taken in the literature on Cellular Automata and shows how to enhance a known self-replicating expression with the ability to apply an arbitary function (program) to arbitrary input, thus achieving self-replication *and* computation.

This self-replicating expression does not have a normal form and normal order reduction, while achieving self-replication, does not achieve application of the program to the data. One strategy for addressing this issue is to use a hybrid of applicative order reduction and normal order reduction. Another strategy is to use parallel reduction. Both approaches achieve self-replication and program application. The hybrid strategy is less appealing because it requires an extension to the basic language of the Lambda Calculus to indicate when to use applicative order reduction. Parallel reduction is closer to the computation in Cellular Automata which is massively parallel. Both these approaches have been implemented in a Lambda Calculus interpreter.

Several interesting variations on the original self-replicating expression are presented. These variants show how to achieve self-replication while also applying a program to a fixed point, keeping a list of the results of program applications, applying a program to a list of input, and mapping a program across a list of input. Naturally, other variants are easy to construct.

The appeal of this work over the work in Cellular Automata is that the Lambda Calculus is so much simpler to work with and it is easier to construct self-replicating expressions which also apply a program to some input where the program is actually capable of useful computation. This simplicity and flexibility is shown by the several variations of the original self-replicating expression that manipulate the program and input data in different ways.

5.1 Future Work

This paper essentially only examines one self-replicating structure with the ability to apply a program to input and concerns itself with the problem of this structure's lack of normal form. The variant self-replicating expressions are variants on a common "stem" expression.

Further research will explore other self-replicating structures. The self-replicating expression discussed in this paper is trivially self-replicating in the sense that the original "program" is encoded in the expression as a parameter. Discovery of non-trivial self-replicating expressions, particularly those with the ability to apply a program to some input, would be very interesting. Of particular interest would be an expression that achieves some level of self-replication with program application using only normal order reduction. An experiment similar to McKay and

Essam's to evolve self-replicating expressions in the Lambda Calculus would be very interesting. The self-replicating expression discussed in this paper, when reduced using parallel reduction, does not replicate its original form, but after a few reduction steps achieves (or evolves into) a self-replicating form. Further investigation into similarly evolving patterns and the number of reduction steps in the evolution into a self-replicating form is warranted.

The computation of a cellular automaton is defined by its starting pattern and also by the state transition rules. The analogy to the Lambda Calculus is that the Lambda Calculus expression is the starting pattern and the reduction rules are the state transition rules. Experimenting with different reduction rules for the Lambda Calculus would be analagous to experimenting with different state transition rules for cellular automata.

This paper considers self-replicating expressions in the untyped Lambda Calculus. In the typed Lambda Calculus, these self-replicating expressions are ill-typed, as, indeed, is the fixed-point Y combinator. This is due to the application of x to itself, producing an infinite type. Investigating self-replicating expressions in the typed Lambda Calculus is both challenging and interesting work.

Finally, an interesting avenue of research would be to explore self-similarity in Lambda Calculus expressions *vis-á-vis* fractal patterns, where the emphasis is not on complete self-replication but on recurring patterns of interesting sub-expressions.[4]

Ackowledgements

Thanks to the anonymous referees, whose comments have improved this paper.

References

Barendregt, H. P. (1984), *The Lambda Calculus—Its Syntax and Semantics (2nd Ed.)*, North-Holland.

Langton, C. G. (1984), 'Self-reproduction in cellular automata', *Physica D* **10**. 135-144.

Langton, C. G. (1986), 'Studying artificial life with cellular automata', *Physica D* **22**. 120-140.

Larkin, J. D. (2003), 'Self-replicating Lambda Calculus expressions'. Honours Thesis, School of IT, Bond University, Australia.

McKay, R. & Essam, D. (2001), Evolving self-reproducing functional programs, *in* 'Proceedings of the Inaugural Workshop on Artificial Life AL'01'. Adelaide, Australia.

Mitchell, M. (1996), Computation in cellular automata, *in* G. et. al., ed., 'Non-standard computation', Weinheim: Wiley-VCH.

Perrier, J.-Y., Sipper, M. & Zahnd, J. (1996), 'Toward a viable, self-reproducing universal computer', *Physica D* **97**. 335-352.

Peyton-Jones, S. (1987), *The Implementation of Functional Programming Languages*, Prentice Hall International.

Tempesti, G. (1995), A new self-reproducing cellular automaton capable of construction and computation, *in* 'European Conference on Artificial Life', pp. 555–563.

Thompson, G. P. (n.d.), 'The Quine Page'. *http://www.nyx.net/~gthompso/quine.htm

von Neumann, J. (1966), *Theory of Self-reproducing Automata*, University of Illinois Press, Illinois. Edited and completed by A. W. Burks.

[4] Thanks to Dr. Robert Barta for this idea.