Communicating Process Architectures 2015
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2015

Lambda Calculus in Core Aldwych

Matthew Huntbach
School of Electronic Engineering and Computer Science
Queen Mary University of London, UK

Abstract. Core Aldwych is a simple model for concurrent computation, involving the concept of agents which communicate through shared variables. Each variable will have exactly one agent that can write to it, and its value can never be changed once written, but a value can contain further variables which are written to later. A key aspect is that the reader of a value may become the writer of variables in it. In this paper we show how this model can be used to encode lambda calculus. Individual function applications can be explicitly encoded as lazy or not, as required. We then show how this encoding can be extended to cover functions which manipulate mutable variables, but with the underlying Core Aldwych implementation still using only immutable variables. The ordering of function applications then becomes an issue, with Core Aldwych able to model either the enforcement of an ordering or the retention of indeterminate ordering, which allows parallel execution.

Keywords. Computational model, single-assignment variables, single-writer variables, lambda calculus, lazy evaluation, mutable variables, multi-agent systems.

Introduction

This paper moves on from our previous work on establishing a core language for concurrent programming. The core language has a simple operational model, and can be used as a real programming language (an implementation exists). The examples we give are executable code, although in an abstract form, we do not discuss how to map it onto actual computer architecture. The principle is that programs should be naturally concurrent, with sequentiality only when a necessary part of what is being coded. One of the aims of our model is to expose the ways in which parallel execution could take place.

The model is based on the idea that computation should be about communicating agents which proceed at their own pace without a central co-ordinator or a central clock, an agent suspending only when it needs information from another agent. It is necessarily non-determinate, as an agent which has alternative paths to follow should be able to choose one of them when it receives sufficient information to make that feasible without having to wait to see if other paths become feasible. A model that can be described precisely needs a strict limit on possible communication, in ours that is done by all communication through variables which may be shared, but always with only one agent having write access. In addition, our model has immutable variables, thus avoiding the complexity which stems from multiple agents making multiple changes to shared variables.

So a key aim is to give something that has predictable and provable behaviour. However, directly programming in such a language is verbose, and may seem limited in its capabilities. One aspect is that it does not have the higher order capabilities seen as an important part of the appeal of functional languages, another is that modeling the real world would seem to need mutable variables as the real world is mutable. We show in this paper how both of these can be represented in our core language.

1. Back Communication and Linear Variables

In our previous papers we first described a general model for concurrent computation using the concept of single-writer single-assignment variables [1] and then showed how this could be expressed in a graphical format to give a more intuitive feel for how it worked [2]. The history of this model is that it derived from attempts to put a syntactic sugar on concurrent logic programming, building on identification of common patterns used when writing code directly in concurrent logic languages, in particular putting an object-oriented style syntax over the logic layer [3]. From this a very simple model of computation emerged, with the key idea of having variables which were guaranteed exactly one writer and could never be re-written. Although the value of this "logic variable" [4] concept had been previously identified, and the suggestion of read-and-write modes [5] on logic variables had been made, this tended to be seen as a convenience somewhat at odds with logic programming idealism. In our work we extended modes from being associated just with predicate arguments to being associated also with the tuples to which logic variables are bound. We also broke down the complex unification concept of logic programming to explicit matching and assignment of each variable.

A necessary part of this was to distinguish between linear and non-linear variables [6]. In our model, every variable must have exactly one writer, however, a variable designated as linear must also have exactly one reader. The reason for this is the concept of "back communication" [7], a feature of concurrent logic programming which distinguishes it from first order functional programming. In functional programming a computation will create a structure and set up computations to construct the components of that structure. Back communication means setting up a structure which may contain components which are supplied by a computation which accepts the structure: that is the reader of the structure becomes the writer of parts of it, and the writer of the structure becomes a reader of those parts. We have added the requirement that if back communication is to take place, it can only be when it can be guaranteed there is exactly one reader for the structure, thus maintaining the single writer property for the variables inside that structure. So, in our model, computations explicitly assign values to variables, and may only do so to variables to which they have write access. A value is a structure which may contain further variables, and the assignment indicates whether the assigning computation (the writer) is to be the reader or writer of each of those variables. Only a variable designated as linear may be assigned a structure which contains variables where the read/write mode is reversed, and only a variable which is designated as linear may be assigned a structure which contains any further variables designated as linear.

With conventional functional programming no distinction is made between a value and a variable which refers to it, but a distinction is introduced in some forms of concurrent functional programming with the concept of a "future" [8], meaning in effect a variable which is currently unassigned as another computation is its writer and has not yet written to it. In our model the explicit use of variables is needed to enable the linear/non-linear distinction to be made and to facilitate back-communication, but also because every variable works in effect as a future but without explicit future handling syntax.

We have noted here the flexibility given by back communication in the logic programming model, but this is countered by the flexibility given by higher order programming in the functional model. Functional programming makes no distinction between the values held by variables and functions, whereas our model shows its logic programming origins in the way that variables can only hold structures, they cannot hold references to computations. There has been a revived interest in functional programming in recent years, in part due to a realisation that its model of computation based on immutable

structures is a solution to many of the problems of large scale programming, particularly when combined with concurrency, stemming from mutable values [9]. However, the ability to write generalised code using higher order features also contributes, as it fits closely with modern principles of good design structure. Our model has the immutability, and in the rest of this paper we shall show that it can model the features of higher order functional programming, so indicating that an explicit higher order model is no needed to provide this flexibility.

We have retained the name "Core Aldwych" for our model, but the emphasis here is on it as a model for general concurrent computation rather than for any specific programming language. It gives a model of a computational structure as executable code, but it is not our intention to promote it as a programming language for direct human use. Higher level forms can be used as a shorthand to give less verbose code [3], but at the expense of having to show how the higher level forms translate to the operational model we use here. Our original Aldwych language compiled into the concurrent logic programming language Strand [10] (hence its name from famous London streets: Aldwych turns into Strand) with the mode concept originating in Parlog [7]. The list notation we use (summarised in section 2.7) is that established as a convention in the logic programming language Prolog [11], which was carried into the concurrent logic languages. The establishment of a guaranteed mode and single writer for every variable through the use of linearity is our own contribution in the logic programming context.

2. The Core Aldwych Model

2.1 Agents and Variables

In the Core Aldwych model we have a network of agents, and a set of variables through which they communicate. An agent has read access to 0 or more variables and write access to 0 or more variables. For every variable there must be exactly one agent which has write access to it. For every linear variable there must be exactly one agent which has read access to it. For any non-linear variables, any number of agents may have read access to it, including 0. It is possible for an agent to have both read and write access to a variable, and these count as separate accesses, so if any agent has both read and write access to a linear variable no other agent can have access to it. A variable may only be written to once, prior to that its value is undefined. An agent which is a reader of a variable may pass the variable's value as the value to be written to another variable of which it is the writer while that value is still undefined, this is termed an "alias". An undefined variable only causes a reader agent to suspend if that reader agent requires its value in the guard to one of its possible commitments (discussed further in section 3).

2.2 Primitive Agents (Assignment, Back-communication and Aliasing)

An agent may be simply an unguarded assignment or alias. An assignment is written textually in the form:

$$x=tag(y_1,\ldots,y_n)$$

representing an assignment of the tuple $tag(y_1,\ldots,y_n)$ to the variable x. This has write access to the variable x and read access to each of the variables y_1 to y_n, which are its input variables. An assignment with back-communication takes the form:

$$X=tag(Y_1,\ldots,Y_n)->(Z_1,\ldots,Z_m)$$

where Z_1 to Z_m represent the back-communication variables. This has write access to the variable X and the variables Z_1 to Z_m and read access to each of the variables Y_1 to Y_n. We

employ the convention that linear variables are shown with initial upper case letters, and non-linear variables are show with initial lower case letters. Although we have shown all the variables in the back-communication structure as linear, this is not a requirement. The tag of an agent is a string, and can be the empty string. The `(` and `)` surrounding the back-communication variables may be omitted if there is just one output variable. An alias takes the form:

```
x<-y
```

and always has exactly one variable with read access, `y` in this case, and exactly one with write access, `x` in this case. Both assignment and aliasing pass the read and write access to variables to other processes. Aliasing passes read access to `y` to the reader of `x` and write access to `x` to the writer of `y`. With assignment, read access to the tuple's input variables and write access to its back communication variables is passed to the reader of the variable being assigned.

2.3 Process Agents

An agent may be a process, which is a set of rules together with the variables to which it has read and write access. An example is given in section 3.1, showing how its is similar to an `ALT` construct in occam. A rule has a left-hand side (lhs) and a right-hand side (rhs), the lhs is like a guard. A rhs is itself an agent, and can take any agent form. The lhs consists of a set of matches, where each match takes the same textual form as an assignment. The variable which is being matched, `x` or `X` above, must be either one of the variables to which the process has read access, or one of the read access variables in another of the matches for the same rule. No variable may be matched more than once in the same rule. Every variable to which an agent has write access must be in a write access position in the rhs of each rule, and any variable in write access position in a match on the lhs must be in a write access position on the rhs. Every linear variable to which an agent has read access must occur either as a variable being matched on the lhs of each rule or once in a read position on the rhs, but not both. Every linear variable in a read access position in a match in the lhs must occur either as a variable being matched in another match on the lhs or once in a read position on the rhs, but not both.

2.4 Compound Agents

An agent may take the form of a compound of agent, which is a collection of agents. The collection may contain additional variables which are neither read nor write variables of the overall agent so long as each is in a write position, and for a linear variable also in a read position, in the internal agents. Although variables are given names for textual display of Core Aldwych code, we can use a purely graphical notation without explicit names. As there are no explicit names, there is no notion of name capture. For convenience we can have named templates for sets of rules, but this is just a representation convenience as well.

2.5 Recursive Agents

For fully anonymous Core Aldwych, we have a recursive agent as a form of process agent. The reason for this is to dispense with what would otherwise be a global aspect, that is, a global set of names for patterns of rules where an agent could be an instantiation of a named pattern.

An agent on the rhs side of a rule may be a recursive agent, in which case it has the same number of read and write access variables as its enclosing process and is treated as having identical rules. As an agent on the rhs could also be a process which is not recursive

so it has its own set of rules, we also have agents which are recursive not to their immediately enclosing set of rules but to a set of rules which encloses that set. This is discussed briefly in section 6.

2.6 The World is an Agent

With anonymous Core Aldwych there is no aspect of the model which involves global co-ordination. The model does not require any additional aspects to model interaction with the world, since the world can be treated as just another agent. Agents can interact with anything so long as what they are interacting with has a Core Aldwych interface consisting of a set of variables to which it has write access and a set to which it has read access.

2.7 Notation Shorthands

For convenience, we introduce the following shorthands. If a tuple is written in the place of a variable inside another tuple, that is a shorthand for an intermediate variable with an assignment, so `X=tag(pair(Y,Z),T)` is shorthand for the two assignments `X=tag(P,T), P=pair(Y,Z)`. We use the list notation of standard Prolog [11], where `X=[Head|Tail]` is equivalent to `X=cons(Head,Tail)`. Also `X=[V1,V2|Tail]` is a further shorthand for `X=[V1|[V2|Tail]]` which in turn is shorthand for `X=cons(V1,Rest), Rest=cons(V2,Tail)`. When the | is omitted it indicates a final `cons(Head,Tail)` structure where `Tail` is assigned the empty list, so `X=[V1,V2]` is shorthand for `X=cons(V1,Rest), Rest=cons(V2,Tail), Tail=[]`. The symbol `[]` represents the empty list in isolation, it is a tuple with no input or back-communication variables. These shorthands apply to both matches on the lhs of rules and assignments on the rhs.

2.8 Operational Semantics

The operational aspects of Core Aldwych can be summarised as follows (see our previous paper [2] for more detail). If a process has read access to a variable, and the agent which has write access to the variable is an assignment, any match on the lhs of its rules which matches the assignment is removed. This may result in a rule having an empty lhs. If a process has a rule with an empty lhs, it may "commit", meaning that the process is replaced by the agent which forms its rhs. Assignments on the rhs are then released to make further matches with other processes. If more than one rule has an empty lhs, an indeterminate choice is made between them, similar to the `ALT` construct in occam. Once a commitment is made, it may not be reversed. If a process is able to commit, it must eventually commit, but there may be an arbitrary amount of time between at least one rule acquiring an empty lhs and actual commitment, which is the principle of unbounded non-determinism used in the Actors model of computation [12]. If another rule acquires an empty lhs in this time, there is no requirement that commitment is to the first rule to acquire an empty lhs.

3. Streams and Continuations in Core Aldwych

3.1 Representation of Streams as Merged Lists

The assignment of a variable `S=[Mess|S1]` can be considered as sending a message, which is whatever `Mess` is assigned to, on a stream represented by `S`, with the variable `S1` taking over representation of that stream. There is no requirement that `Mess` is already

assigned to do this, the process which reads S can take the message with a matching match, and commit before Mess is assigned if it does not need the value of the message. Using this, we can write a process which merges two streams as:

```
(Left,Right)->Stream
{
 Left=[] || Stream<-Right;
 Right=[] || Stream<-Left;
 Left=[Mess|Left1] || *(Left1,Right)->Stream1, Stream=[Mess|Stream1];
 Right=[Mess|Right1] || *(Left,Right1)->Stream1, Stream=[Mess|Stream1];
}
```

The || separates the lhs from the rhs of a rule, and ; indicates the end of a rule. The * indicates a recursive agent. The first two rules represent the case when one of the input streams is closed by being set by its writer to []. Messages from the other stream are then forwarded, which can be done by setting Stream to alias it rather than separately reading and forwarding each message.

If the merge process given above is combined with the assignment Left=[V|ContL], the result is that the combination evaluates to

```
Stream=[V|Stream1], *(ContL,Right)->Stream1
```

and the recursive call *(ContL,Right)->Stream1 is replaced by the same set of rules with Left replaced by ContL and Stream replaced by Stream1. The release of the assignment Stream=[V|Stream1] means the process which has read access to Stream may similarly commit and release further assignments and so on.

If the assignment Right=[U|ContR] were available, the process could commit using the fourth rule instead of the third rule, meaning the reader of Stream gets the message in U rather than the message in V. This shows how non-determinacy is a natural aspect of the Core Aldwych model, there is no special syntax for it. In Core Aldwych a commitment can be made without waiting to see if another commitment could have been possible. If both Left=[V|ContL] and Right=[U|ContR] were available, a commitment to either rule is possible. We have not added a mechanism for expressing a priority since this introduces a considerable extra complexity to the semantics of the language. For similar reasons there is no equivalent to an occam SEQ structure, as that would require mechanisms to maintain dependence between processes beyond the variable reading and writing described here.

3.2 Core Aldwych and CSP: Similarities and Differences

Thinking of programming in what was originally a concurrent logic language in terms of streams on which messages are sent brings it close to the CSP model [13]. In this paper we show how other core computational models, in particular lambda calculus [14], can also be translated to Core Aldwych, thus establishing it as a flexible model for general computation with concurrency as a natural aspect rather than an added-on extra to a purely sequential core as still tends to be the case with conventional programming languages [15].

Note, however, that whereas CSP is synchronous, Core Aldwych is asynchronous. When a process in Core Aldwych commits to a rule containing the assignment S=[Mess|S1] with S an output variable of the process, there is no suspension waiting for the reader of S to commit to a rule with a lhs containing the match S=[Mess|S1]. It could be that the reader of S is suspended waiting for that assignment, but it could be that it is suspended waiting for the assignment of another variable and cannot read S until that happens. As with the merge example above, it could be suspended on more than one variable, so it could commit on the assignment of another variable whether or not S is

assigned. Another possible pattern is that it is suspended on `S` and another variable and cannot commit until both are assigned, this is shown in an alternative stream merger:

```
(Left,Right)->Stream
{
 Left=[] || Stream<-Right;
 Right=[] || Stream<-Left;
 Left=[MessL|Left1], Right=[MessR|Right1]  ||
           *(Left1,Right)->Stream1, Stream=[MessL,MessR|Stream1];
}
```

These process rules would give a merged stream in which the messages from the two streams are strictly alternated, and a message from one stream cannot be sent on until either a message has also been received on the other stream, or the other stream has been closed.

4. Representation of Expressions and Functions

4.1 The Continuation Passing Style of Programming

The use of the `*` notation for a recursive call encourages the idea of seeing the call as a continuation of the overall process. When a process commits to a rule it terminates by creating a compound agent from its rhs. However, if the rhs includes a recursive call, we can see that call as a continuation of the same process. We can see the arguments to the recursive call as a mutable state. Setting the head of an output list to a message, with the tail in the same position in the recursive call writing the rest, is the process sending a message. Thinking in this way is the Continuation Passing Style of programming [16], which is well established in functional programming. This technique was the basis on which object-oriented programming was previously modelled in concurrent logic languages [17], although as the emphasis was on objects as concurrently communicating entities rather than aspects such as inheritance it resembled the Actors model of computation [12].

In section 4 we show how the technique used for representing mutable objects in concurrent logic languages can be used as a general computational model, leading up to its use to represent lambda calculus. The aim is to establish a standard computational model which can be used to establish clear semantics for programming languages in general, including concurrent aspects. The point of representing lambda calculus is that it has long been established as a computational model for this purpose itself [14], but it is an insufficient model for handling concurrency and interaction. We then show how various different ways in which a single lambda calculus expression could be interpreted in a concurrent and interactive environment can all be represented in Core Aldwych.

4.2 Representation of an Entity by an Input Stream

To use Core Aldwych as a general model for computation, we represent computational entities by an agent with one linear input variable. This input variable represents a stream of messages sent to the entity, in the form as given above. If we wanted to model the idea of separate "views" of an object, that could be done through an agent with several linear input variables, one for each view, but we will not consider that possibility further here. If an entity contains references to other computational entities, those references are represented by a linear output variable for each of them.

This representation is counter-intuitive. A general notion of computational entities is that one is created by taking some inputs, putting them together, and producing an output representing the entity. Here, to represent an entity which contains references to other

entities, we take some output streams which represent those references. We put them together and construct an agent writing to them to represent the entity. The agent has a single input stream, and this input stream is now the access to the representation of the entity to other entities. So, it is the reverse of the general notion, as here we take some outputs, put them together and end up with an input representing the entity. If many agents need shared access to one particular agent, each will output its own stream of message to it, and the streams are merged using the process given in section 3.1 into the single stream which is read by that particular agent.

The syntactic sugar we introduced in earlier papers leading to the language we called just "Aldwych" [3] hid this reversal and hid the explicit merger of streams used to create multiple references to objects. Getting to grips with this reversal of modes is the key to understanding how Core Aldwych works as a representation.

4.3 Representation of Constants

We represent a constant by an agent which takes a stream of messages, for each just returning the value of the constant. From now on we will in most cases give names to patterns of processes rather than use the anonymous notation shown above, so with name `constant` this is:

```
constant(S,val)
{
 S=empty() ||;
 S=[Mess|S1], Mess=()->ret || ret<-val, constant(S1,val);
}
```

The first rule shows how an entity is removed when the stream of messages to it is an empty list. The process representing the constant reads the tuple `empty()` on its input variable `S`, and as `empty()` is a tuple with no linear input variables and no output variables nothing else can be done, so the rhs is empty. In the rest of this paper, following the shorthand given in section 2.7, we will use `[]` where here we used `empty()`.

The second rule shows the double variable matching explicitly to indicate more clearly the underlying operational model, but in our further examples we will employ the syntactic sugar which means the lhs side of the second rule would be written `S=[()->ret|S1]`. Here `()->ret` is a tuple with no input variables and one back-communication variables, that is $\texttt{tag(Y}_1\texttt{,…,Y}_n\texttt{)->(Z}_1\texttt{,…,Z}_m\texttt{)}$ as in section 2.2 with `n` having value 0, `m` having value 1 and the empty string as its tag. The alias `ret<-val` sets `ret` to whatever value `val` was or will be given by its writer. There is no requirement that `val` has already been assigned a value for this rule to operate, since though the constant process is a reader of `val`, there is nothing on the lhs of its rules which checks `val` for any value.

4.4 Representation of Expressions

Here is how the arithmetic expression `z=x+y` is represented:

```
addex(Z)->(X,Y)
{
 || X=[()->xval], Y=[()->yval], zval<-xval+yval, constant(Z,zval);
}
```

This is a process agent with one input stream `Z` and two output streams `X` and `Y`. The different font for `x`, `y` and `z` here indicates an expression in some language being represented in Core Aldwych rather than a Core Aldwych expression. The intention is that these are functional style variables, so it is a definition of the value of `z` and not an

assignment of a mutable variable. It is a process with one rule that has no matches, meaning that wherever it occurs it can be immediately replaced by the rhs of the rule. Core Aldwych incorporates standard arithmetic expressions, so `zval<-xval+yval` is just setting `zval` to the sum of `xval` and `yval`, the single-assignment principle applies, so this is the one write position of `zval`.

The reversal of modes technique can be seen clearly here. The streams `X`, `Y` and `Z` represent the variables `x`, `y` and `z` respectively. The representation of a calculation which takes `x` and `y` as input and gives `z` as output is a process which takes `Z` as an input stream and produces `X` and `Y` as output streams.

The wrapping of values in processes here has similarities to the functional programming style of wrapping values in monads [18]. The value `zval` has to be wrapped in a process rather than returned directly for the stream `Z` to represent the output in order to fit in with the principle of all entities represented by streams with each individual request for its value represented by a separate message. Clearly it is not efficient to represent a constant value in this way, however, as noted in section 4.1 the point of this work is to establish clear semantics, not to give an efficient implementation.

The unwrapping of a value from a process is done by sending a message to the process in the form `()->value`. That is done only when the actual value is needed, as for arithmetic calculations. It can be done once for multiple uses of the same constant, for example `w=x*x` is represented by:

```
squareex(W)->X
{
 || X=[()->xval], wval<-xval*xval, constant(W,wval);
}
```

The output streams are closed off in these expression representations by setting them to lists of a fixed length, but in a wider context they would be merged with other streams representing aliases, so `z=x+y`, `w=x*x` with further uses of `x` is represented by:

```
addex(Z)->(X1,Y), squareex(W)->X2, merge(X1,X2)->X3, merge(X3,X4)->X
```

where `X4` is the representation of `x` following the expressions. Another use of `x` would be represented by `X5` being output from the agent representing it, `merge(X5,X6)->X4` being added, and `X6` is then the representation for further uses of `x`.

As a minor point, arithmetic expressions in Core Aldwych are written using the alias symbol rather than then assignment symbol, so `zval<-xval+yval` rather than `zval=xval+yval` and so on, because the right hand side of an assignment is always a tuple. If we had `zval=xval+yva` , it would be interpreted as setting `zval` to a tuple with tag `+` and two variables set to the values of `xval` and `yval`.

4.5 Representation of Functions

Section 4.4 shows one-off expressions, but for a function we need an agent which can be used repeatedly. Here is how a square function is represented, it is the equivalent of λx.x*x as a lambda expression:

```
square(S)
{
 S=[] || ;
 S=[(Res)->X|S1] || X=[()->xval], wval<-xval*xval,
                    constant(Res,wval), square(S1);
}
```

In general a function is represented by a process which reads a stream of messages of the form `(Res)->Arg`. In this example we have used `X` rather than `Arg`, as it represents the argument to the function which was named `x`. This is a tuple with back communication as described in section 2.2 with an empty string as its tag and `n` and `m` both 1, that is one input variable and one back-communication variable. We can see the mode reversal again: the argument to the function is represented by an output of a stream of messages `X` to the agent representing the argument `x`, the result of applying the function to the argument is represented by an input of a stream of messages `Res` to the agent representing the result. The recursion enables the function to be used again. Processes which use a function have as their representation of it an output stream on which messages are sent, with streams from several processes which access one function merged into the one which it reads.

To apply a function to an expression, we need to send a `(Res)->Arg` message on a stream read by the process which represents the function. In this message, the output stream `Arg` represents the argument to the function call and the input stream `Res` represents the results of the function call. So the functional programming style expression `z=f x` where `f` is a function, is represented by

```
apply(Z)->(F,X)
{
 || F=[(Z)->X];
}
```

This sets the output stream `F` to a stream with a single message `(Z)->X`. As previously, if we wish to make further use of `f` and `x` the streams output here need to be merged with streams representing the further use. The output stream `X` is not closed off instead the reader of `F` will become the writer of `X`. So if we had

```
apply(Z)->(F,X), square(F)
```

the process `square(F)` will set `X` to `[()->xval]`. That is, `F` is a stream of messages representing calls to the square function read by the process given by the rules named `square`, this stream could be merged with other streams from other processes which access the same square function. The process which reads `F` reads a `(Z)->X` message from it, sends a `()->xval` message on the stream `X` to get the value of `x`, calculates the square of that value, and sets up a process given by the pattern of rules named `constant` making that the reader of the stream `Z` with the calculated value in `wval` as its non-linear input.

A free variable in a lambda expression is represented by an output stream, so `λx.x+a` with the free variable `a` is represented by:

```
add1(S)->A
{
 S=[] || A=[];
 S=[(Res)->X|S1] || X=[()->xval], A1=[()->aval],
                    rval<-xval+aval, constant(Res,rval),
                    merge(A1,A2)->A, add1(S1)->A2;
}
```

The free variable `a` is shared between its use in the expression `x+a` of the function and its use in further applications of the lambda expression, so the two separate streams `A1` and `A2` representing these uses are merged into one which is output to be read by whatever process represents `a`. When no further uses are made of the function, represented by its input stream set to the empty list, it is necessary to set the variable representing `a` to the empty list to show that it will not be used further in the function. The output stream `A` could be merged with other streams representing other processes sharing access to `a`.

The standard functional programming idea of currying can be used to give functions

with multiple arguments, so λy.λx.x+y is represented, using `add1` as above, by:

```
add(S)
{
 S=[] || ;
 S=[(Res)->Y|S1] || add1(Res)->Y, add(S1);
}
```

More generally, if `E` with `exp(E)->(A,`V_1`,…,`V_n`)` is the representation of an expression exp, which takes a single argument `A` with $V_1,\ldots,V_n$ representing its free variables, then λx.exp is represented by `S` with:

```
lambda(S)->(V1,…,Vn)
{
 S=[] || V1=[], …, Vn=[];
 S=[(Res)->X|S1] || exp(Res)->(X,V11,…,V1n),
                    lambda(S1)->(V21,…,V2n),
                    merge(V11,V21)->V1, … , merge(V1n,V2n)->Vn;
}
```

4.6 The Y combinator

The fixed point operator or Y-combinator is a function Y which has the property that for any f, the expression Y f evaluates to f (Y f). It is used to provide recursion in lambda calculus while maintaining the principle of anonymous functions. A fixed point operator like this can easily be defined in Core Aldwych, here it is:

```
fixedpoint(S)
{
 S=[] || ;
 S=[(Res)->F|S1] || F=[(R1)->F1], merge(Res,F1)->R1, fixedpoint(S1);
}
```

If we have `fixedpoint(Y), apply(Res)->(Y,F)` this will evaluate to the assignment and merge process `F=[(R1)->F1], merge(Res,F1)->R1`. Here `Res`, which is the stream of messages directed to the representation of Y f, is merged with `F1` into one stream, `R1`. This means `R1` represents applying what `F` represents to what `F1` represents. `F` is the representation of f and the merger means `F1` and `Res` both represent Y f which through `R1` is also f (Y f).

4.7 Lazy Evaluation

The representation of function application given above causes the message `(Res)->Arg` to be sent to the agent representing the function. As described in section 4.5, if the agent were a `square` process this would lead to `Arg` being set to `[()->xval]`, the square of `xval` being calculated and a `constant` process being set up to wrap it. However, there is the idea of lazy evaluation, which means carrying out the work of a computation is delayed until the result is needed. If the result is never needed, the work need not be carried out at all. This can be modelled with an alternative form of `apply`:

```
lazyapply(Res)->(F,X)
{
 Res=[] || F=[], X=[];
 Res=[Mess|Res1] || F=[(Z)->X], Z=[Mess|Res1];
}
```

If we have `constant(T,3), square(S), lazyapply(Res)->(S,T)`, which sets `Res` as the stream of messages to the representation of square 3 applied lazily, then if `Res` is set to `[]`, meaning no use is made of the result, `T` and `S` will be set to `[]`, closing the `constant` and `square` processes with no calculation taking place. If one message is received, the calculation will take place and all further messages are directed to the result, so the calculation is only done once, which is correct lazy evaluation behaviour.

So, we have shown here that not only can Core Aldwych be used to represent lambda calculus, it can also represent it in a way in which there is a programmable run-time choice between lazy and eager evaluation in any function application. It is not an efficient representation, and the representation of constants as a stream of messages each of which just returns the same result may seem in particular to be unnecessarily verbose. However, an important point is that the representation would still work even if what we have called constants were not constant.

5. Mutability

5.1 Interaction with Changing Values

Consider that if instead of the `constant` process of section 4.3 we had the following:

```
count(S,val)
{
 S=[] ||;
 S=[Mess|S1], Mess=()->ret || ret<-val, val1<-val+1, count(S1,val1);
}
```

The representation of expression and functions as given above would still work, but if `S=[()->val1,()->val2]` were combined with a reader of `S` that was a `count` process rather than a `constant` process, `val1` and `val2` would no longer have the same value when assigned. We now have the incorporation of mutable values into what was a lambda calculus framework. As anything can be incorporated into a Core Aldwych universe so long as it is given a Core Aldwych interface (interaction through single assignment variables), the mutability need not itself be defined by Core Aldwych as it is in `count` above, it could link to some external factor. Note that when it is defined in Core Aldwych, it is a representation of mutability and not actual mutability. It is still based on a universe where agents interact through single-assignment, that is immutable variables.

Once we have accepted the possibilities of this representation of mutability, possible variations in the representation are opened. If we have multiple uses of a mutable value, one variation is to obtain a fixed value once through one message, as we did in `square` in section 4.5, another is to make multiple requests for the value. The version of square which did it this way would be:

```
squarem(S)
{
 S=[] || ;
 S=[(Res)->X|S1] || X=[()->xval1,()->xval2], wval<-xval1*xval2,
                    constant(Res,wval), squarem(S1);
}
```

Although it makes no difference here, under other circumstances changing the order in which the `()->val` messages are sent would create more versions. Lambda calculus does not take into account order of function application, our representation does, and it is essential when we consider an agent as interacting rather than just calculating a final value.

5.2 Representation of Mutable Variables

What we saw with `count` is the use of Core Aldwych to represent agents with a state. This originates from the insight that an object with state can be represented in concurrent logic programming by a clause with a recursive call, the change in arguments to the recursive call from the enclosing call representing the change in state [17]. To consider this further, we can add a representation of mutable variables to our previous representations:

```
var(S)->Val
{
 S=[] || Val=[];
 S=[()->Val1|S1] || Val=[], var(S1)->Val1;
 S=[(Val1)|S1] || merge(Val1,Val2)->Val, var(S1)->Val2;
}
```

A call `var(V)->Val` represents the declaration of a variable which is initialised to the expression represented by the reader of the stream `Val`. As previously, `Val` could be merged with other streams to become the input to the representation of one expression. The first rule represents the termination of the variable when no further messages are sent to it, the process must indicate it will send no more messages to the current value of the variable by setting its output stream to `[]`. The second rule, taking the message `()->Val1`, represents the assignment of the variable to a new value given by the output stream `Val1`. As the process no longer accesses its old value, its stream to the old value is terminated by setting it to `[]`, and the continuation of the process has a stream to the new value as its output stream. The third rule, taking the message `(Val1)`, represents obtaining the value stored in the variable, with `Val1` a stream of messages sent to the agent representing that value merged with `Val2` which is the stream of messages coming from further access to the variable.

The process template `var` has no dependency on the type of messages in the streams representing values, so it can be used to represent variables which refer to functions or to other variables as well as constant values. For an example:

```
var(N)->Val1, constant(Val1,3), var(Ptr)->N1, merge(N1,N2)->N
```

represents what would be written in the C language as int n=3; int *ptr=&n with `N2` the stream of messages representing further access to n, and `Ptr` the stream of messages representing access to the variable ptr. Compare this with:

```
var(N)->Val1, constant(Val1,3), var(M)->Val2, merge([(Val2)],N2)->N
```

representing int n=3; int m=n. So ptr is set to refer to the variable n, not its value as m is.

5.3 Call by Value and Call by Reference

Having introduced mutable variables, we can now represent functions whose arguments are variables, and have both call-by-value and call-by-reference forms, and do it within the lambda calculus framework. If we use the lambda process of section 4.5 we get call-by-reference, with the expression of the function having direct get and set access to the variable passed to it. The expression fv, where v is a mutable variable, could be represented by obtaining the value stored in v and applying f to it. If `F` and `V` are streams with readers representing f and v respectively, that is done by

```
apply(Res)->(F,Val),V=[(Val)]
```

where `Res` is the stream representing the result and `apply` is as in section 4.5. It also would work using `lazyapply` from section 4.7.

However, more in line with standard procedural programming, if exp is an expression

containing use of a mutable variable m, and `exp(Res)->M` represents it with `M` the output of a stream of messages to the representation of `M`, then what we might write as λ*m.exp to indicate that m is a variable passed by value is represented by:

```
vlambda(S)
{
 S=[] || ;
 S=[(Res)->M|S1] || M=[(Val)], var(M1)->Val, exp(Res)->M1, vlambda(S1);
}
```

That is, a local variable is set up and initialised to the value of the argument passed, and here that argument is passed as a variable so its value is explicitly obtained. Free variables in exp would be represented, as previously, by additional output streams.

5.4 Locked and Ordered Access

Below is the representation of a function which takes a variable set to an integer and passed using call-by-reference, and changes its value by adding n to it. It could be written as λ&x.x:=x+n. The free variable n is represented by an output stream assumed to have a `constant` process as in section 4.3 as its reader. So the value of `n` is obtained directly by sending the message `()->nval`, whereas the value of x as a mutable variable has to be obtained in two stages, first by sending the message `(Val1)`, to get the value representation in `Val1`, then by sending `()->xval1` to get the actual value in `xval1`.

```
addN(S)->N
{
 S=[] || N=[];
 S=[(Res)->X|S1] || X=[(Val1),()->Val2], merge(N1,N2)->N,
                    Val1=[()->xval1], N1=[()->nval],
                    sum<-xval1+nval, constant(Sum,sum),
                    merge(Res,Val2)->Sum, addN(S1)->N2;
}
```

The function returns the new value of the variable x as well making the change, so `Res` is the stream representing the return value and `Val2` the stream representing access to the value stored in x through the variable itself. As they are the same value, they are merged. The getter message `(Val1)` then the setter message `()->Val2` are sent in that order to the stream `X` with its reader the process representing the variable x. Now suppose we have the following:

```
constant(0,Z), constant(1,N1), constant(2,N2), var(Z,X),
addN(S1)->N1, addN(S2)->N2, S1=[(Res1)->X1], S2=[(Res2)->X2],
merge(X1,X2)->X3, merge(X3,X4)->X
```

This represents setting up a variable x with initial value 0, and executing x:=x+1 and x:=x+2 concurrently. The stream `Res1` can be sent `()->val` messages to get the value of x after executing x:=x+1 and the stream `Res2` can be sent `()->val` messages to get the value of x after executing x:=x+2. The stream `X4` is where further getter and setter messages can be sent to the representation of x. What actually happens?

`X1` and `X2` are both set to lists containing two messages, a getter followed by a setter. If we write this as `X1=[(Val11),()->Val21], X2=[(Val12),()->Val22]` we can see that the indeterminate merger of the two streams can lead to several results, modeling the classic concurrent access issue which means x could end up as set to 1, 2 or 3. The merger to `[(Val11),()->Val21,(Val12),()->Val22]` leaves x set to 3, the merger to `[(Val11),(Val12),()->Val21,()->Val22]` leaves x set to 2 and so on. In fact it is worse than this, since `X4` is indeterminately mixed with the result. When our functions

have side effects, the ordering of messages to them matters.

However, this can be managed. We can model sequential access by appending the streams rather than merging them. So, with:

```
constant(0,Z), constant(1,N1), constant(2,N2), var(Z,X),
addN(S1)->N1, addN(S2)->N2, S1=[(Res1)->X1], S2=[(Res2)->X2],
append(X1,X2)->X3, append(X3,X4)->X
```

we represent x:=x+1; x=x+2 executed sequentially, with x left set to 3, and the messages on X4 not dealt with until that is done. Note that unless we have need to retain access to older values of x, Res1 and Res2 here can be set to []. The rules for the append process are:

```
append(Left,Right)->Stream
{
 Left=[] || Stream<-Right;
 Left=[Mess|Left1] || append(Left1,Right)->Stream1,
                      Stream=[Mess|Stream1];
}
```

If E is a stream read by the representation of an expression, then append(E1,E2)->E splits it into two references to the expression, E1 and E2, but with E1 locking it. The messages on E2 will not go through to E until the writer of E1 has closed it by setting the end of the list to []. Other ways of merging streams of messages could be employed, such as the strict alternation between two streams we gave in section 3.2. This enables us to consider and model variations in language behaviour. Use of append enforces sequential handling. Use of the indeterminate merge gives handling of data communications from two agents which would cover the two agents sending communications in parallel. Note that because Core Aldwych is asynchronous, with append(E1,E2)->E the process producing E2 may proceed in parallel with that producing E1 so long as it does not require any back communication from the reader of E. The reader of a Core Aldwych variable cannot enforce synchronization on its writer unless there is back communication and the writer has a dependency on the back communication value. Synchronized interaction can be implemented if required through the use of back communication.

5.5 Lockable Variables

If V is read by a stream representing a mutable variable, then append(V1,V2)->V gives us two streams with a guarantee that messages sent on V2 will only be received after all messages sent on V1 have been received. However, that does not guarantee freedom from interference as it may be that V is indeterminately merged with another stream to give the stream which the actual variable representation reads. To guarantee freedom from interference we need to have the stream appending at the variable representation end. This can be done with a variation of the process structure which represents mutable variables:

```
lockablevar(S)->Val
{
 S=[] || Val=[];
 S=[()->Val1|S1] || Val=[], lockablevar(S1)->Val1;
 S=[(Val1)|S1] || merge(Val1,Val2)->Val, lockablevar(S1)->Val2;
 S=[lock(S1)|S2] || append(S1,S2)->S3, lockablevar(S3)->Val;
}
```

The effect is that if V is read by a stream representing a lockable mutable variable, then sending the message lock(V1) on V means V1 really is a stream which accesses the variable and locks it so no other access is handled until access through V1 is complete.

6. An Extended Example: the Twice Function

For a detailed example, consider the lambda expression λf.λx. f (f x), the twice function. The Core Aldwych representation of this as standard lambda calculus is given below. Note, this is a case where the rhs of a rule has a process which is given by an actual set of rules rather than by a call to an external named set of rules. The rhs of the second rule sets up a recursive call to `twice`, and an anonymous process call which has input `T` and output `F`, with the rules for that process following. As it is anonymous, its own recursive call with input `T1` and output `F1` is given by `*(T1)->F1`.

```
twice(S)
{
 S=[] ||;
 S=[(T)->F|S1] || twice(S1),
                  (T)->F {
                     T=[(Res)->X|T1] || *(T1)->F1,
                                        App1=[(Res1)->X],
                                        App2=[(Res)->Res1],
                                        merge(App1,App2)->F2,
                                        merge(F2,F1)->F;
                     T=[] || F=[];
                  };
}
```

Sending a message `(T)->F` on a stream `S` where the reader of `S` is a `twice` process and the reader of `F` represents a function f results in the reader of `T` representing a function which takes an argument, applies f to it, and then applies f to the result and returns what that gives. This is a simple example of a higher order function, the anonymous inner process is the representation of the function created by applying twice to a function f.

Note that if `twice` itself were anonymous, `twice(S1)` would be written `*(S1)`, and if it had been necessary to have an indirect recursive `twice` with an argument `A` inside the internal anonymous process, that would have been written `**(A)`.

The representation of (twice f) for some f needs to send a stream of messages to the reader of `F`, representing calls of the function f. A call (twice f) x gives the same result as a call f (f x). It is initiated by the process representing (twice f) receiving a message `(Res)->X`, with `X` the stream read by the agent that represents x, and `Res` is the stream which takes messages requesting the value of (twice f) x. There is no dependency on the type of messages sent through `Res`, so they could take the form `()->val` if f is a first order function or `(R)->Arg` if f is also a higher order function.

Two messages are sent to `F`, they are `(Res1)->X` representing the call f x, and `(Res)->Res1` representing the result of the call of f x passed as the argument to a further call of f. The variable `Res1` holds the stream of messages which the agent representing the further call of f sends to the agent representing the first call of f to get the value of the call. However, there is no actual ordering of these calls, the `merge(App1,App2)->F2` means they could come in either order. They may not even come one after the other, as the `merge(F2,F1)->F` merges them indeterminately with messages to f coming from further calls of (twice f) applied to different arguments.

If there are no side effects the order to which calls of a function are made, this makes no difference, so although the streams of messages to the function f are merged indeterminately, the same overall result would be given if they were appended. It may seem odd that the function f may have to handle a call to evaluate f (f x) when it has not yet received the call which would evaluate f x, but all this means is that it sets up what it can to

evaluate f(f x) with a variable representing the value of f x on which are sent messages, and when it receives the call to evaluate f x it sets up the reader of this variable.

However, if f were a function with a side effect it would matter. Here is the representation of one such function:

```
addstore(S)->(Const,Var)
{
 S=[] || Const=[], Var=[];
 S=[(Res)->Arg|S1] || addstore(S1)->(Const1,Var1),
                      Arg=[()->val], Const=[()->cval|Const1],
                      sum<-val+cval, constant(sum,Sum),
                      Var=[()->V|Var1], merge(Res,V)->Sum;
}
```

It adds a constant value to its argument, given by stream `Const`, but also sets a mutable variable, given by stream `Var`, to the sum. If the mutable variable were initially set to 0 and the constant value was 5, evaluating twice f 20 would always give 30, but the variable could be left set either to 25 or 30 depending on the order in which the messages to f were sent. Note that this function representation is already constrained, since the assignment `Var=[()->V|Var1]` means that the setter message `()->V` always comes before setter message from later calls of f. If we wanted to make it a general principle that calls of a function should work so that the side effects of a call always have their effect before side-effects of later calls of the same function, we need to change the way free variables are dealt with in functions so that λx.exp with free variables $V_1,\ldots,V_n$ is represented by S with:

```
lambda(S)->(V1,…,Vn)
{
 S=[] || V1=[], …, Vn=[];
 S=[(Res)->X|S1] || exp(Res)->(X,V11,…,V1n),
                    lambda(S1)->(V21,…,V2n),
                    append(V11,V21)->V1, … , append(V1n,V2n)->Vn;
}
```

that is, the streams to the free variable reader from the function calls are appended rather than merged. However, this does mean that each function call locks the free variables it accesses.

In the representation of twice above, each call of the function f is represented by a separate list of one message, with the lists indeterminately merged so the messages could be passed to the representation of f in either order. If the messages are put into one list, that establishes a fixed order on multiple calls to one function in an expression. If the assumption was that inner calls come first it would result in the twice function being represented by:

```
twice(S)
{
 S=[] ||;
 S=[(T)->F|S1] || twice(S1),
                  (T)->F {
                    T=[(Res)->X|T1] || *(T1)->F1,
                                       App=[(Res1)->X],(Res)->Res1],
                                      append(App,F1)->F;
                    T=[] || F=[];
                  };
}
```

This is an example of where attempting to encode a higher level feature in Core Aldwych reveals an aspect which needs further definition to give an exact behaviour. To put it formally, the Church-Rosser theorem [19] does not apply with interactive computation.

A further variation makes the function calls within the call of the twice function lazy, as covered in section 4.7. This leads to the following representation:

```
twice(S)
{
 S=[] ||;
 S=[(T)->F|S1] || twice(S1),
                  (T)->F {
                    T=[([])->X|T1] || *(T1)->F, X=[];
                    T=[([Mess|More])->X|T1] || *(T1)->F1,
                                       Res=[Mess|More],
                                       App=[(Res1)->X],(Res)->Res1],
                                       append(App,F1)->F;
                    T=[] || F=[];
                  };
}
```

In this case, the distinction between lazy and non-lazy evaluation is well understood. The Core Aldwych encoding supports it being a choice that could be made by annotations on individual function applications [20] rather than fixed in the higher level language.

7. Conclusions and Future Work

We have not introduced any new syntax or operational behaviour into Core Aldwych. The language has the same simple operational semantics as when we first described it. It is no more complex in those terms than other models of computation. It is naturally concurrent and naturally handles interaction. We have not had to introduce explicit mutable cells on top of an immutable model [21]. Our model remains immutable, developing by giving values to variables but never changing those values once they are given. Yet we have shown how mutable structures can be modelled, and followed the consequences. As we have shown, lambda calculus on its own is not a sufficient model under these circumstances because there are many different ways in which even quite a simple lambda expression can be interpreted, with different interpretations having different behaviours. Core Aldwych enables us to model those different interpretations and different behaviours explicitly. We have not had to introduce different variations of our model to capture such things as lazy and non-lazy evaluation, or call by reference against call by value. Our one model can be used for all of these.

This paper has been written informally to give a feel for what can be covered using Core Aldwych. We have concentrated on lambda calculus, as this is generally seen as the core computational model, but we have also covered the core mutable variable aspect of procedural programming. There is not space here to show how Core Aldwych could be used to cover explicit models of concurrent computation, such as pi calculus, but it is hoped some of the techniques given here can indicate how it could be done. In fact we have already used Core Aldwych to cover various process calculi, with questions on exactly how they should operate and possible variations established through the route of considering exactly how how they could be implemented in executable Core Aldwych code. Some feel for this may perhaps be obtained by seeing how the decision on how to join streams in the representation of lambda calculus: indeterminate merge, or append, or something else, leads to quite strong differences in actual operational behaviour.

An important aspect of future work with Core Aldwych is the establishment of a normal form for Core Aldwych agents, which would establish a denotational as opposed to an operational semantics. An aspect of Core Alwych which helps with this is that its evaluation is by its nature partial evaluation. Core Aldwych evaluation handles unbound variables and takes it as far as it can, until it has to suspend because it can go no further without knowing their values. As Core Aldwych rules are themselves Core Aldwych agents we can move from the assignment absorption discussed in our previous paper [2], which partially evaluates an agent to a subset of its original rules to reflect its environment and carry on evaluation inside the rules. The goal is to take it to the point where if two Aldwych agents are identical in operational behaviour they will always partially evaluate to the same normal form. To achieve this we need a composition process so that multi-process agents can be composed into one process. We established the basis for how to do this some time ago [22].

References

[1] M. Huntbach, The Core Language of Aldwych. *Communicating Process Architectures 2007*, pp 51-66, IOS Press, Amsterdam, 2007.

[2] M. Huntbach, A Model for Concurrency Using Single-Writer Single-Assignment Variables. *Communicating Process Architectures 2011*, pp 255-272, IOS Press, Amsterdam, 2011.

[3] M. Huntbach, Features of the Concurrent Programming Language Aldwych. *2003 ACM Symposium on Applied Computing*. ACM, pp 1048-1055, 2003.

[4] S. Haridi et al, Efficient Logic Variables for Distributed Computing. *ACM Trans. on Programming Languages and Systems* 21 (3) pp 569-626, 1999.

[5] K. Ueda, Experiences with Strong Moding in Concurrent Logic/Constraint Programming. *Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, pp 134-153. Springer LNCS 1068, 1996.

[6] U. Reddy, A Typed Foundation for Directional Logic Programming. *Extensions of Logic Programming (ELP'92)*, pp 282-318 Springer LNCS 660, 1993.

[7] S. Gregory, *Parallel Logic Programming in Parlog*. ISBN: 0201192412, Addison-Wesley 1987.

[8] J. Niehren, J. Schwinghammer and G. Smolka, A concurrent lambda calculus with futures. *Theoretical Computer Science* 364 pp 338-356, 2006.

[9] R. Hickey, Are We There Yet? (Keynote speech) *JVM Language Summit 2009* Sun Microsystems `http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey`, 2009.

[10] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*. ISBN:0-13-850587-X, Prentice Hall, 1990.

[11] W.F. Clocksin and C.S. Mellish, *Programming in Prolog: Using the ISO Standard*. Springer, ISBN:978-3-540-00678-7, 2003.

[12] G. Agha and C.Hewitt, Actors: a Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, MIT Press, ISBN:0-262-19264-0, 1987.

[13] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall ISBN:0-13-153271-5, 1985.

[14] P. Landin, A Correspondence between Algol-60 and Church's Lambda-Notation, *Communications of the ACM* 8 (2) pp.89-101, 1965.

[15] D.Lea, Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley, ISBN:0-201-31009-0, 1999.

[16] J.C. Reynolds. The Discovery of Continuations, *Lisp and Symbolic Computation* 6, pp.233-248, 1993.

[17] E.Y. Shapiro and A. Takeuchi, Object Oriented Programming in Concurrent Prolog, *New Generation Computing,* 1, pp 25-48, 1983.

[18] M. Erwig and D. Ren, Monadification of Functional Programming. *Science of Computer Programming* 52 (1) pp 101-129, 2004.

[19] A.Church and J.Rosser Some Properties of Conversion, *Trans. Am. Math Soc*. 39 (3) pp.472-482, 1936.

[20] F.W. Burton, Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs. *ACM Trans. on Programming Languages and Systems* 6 (2) pp 159-174, 1984.

[21] P. van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. MIT Press, ISBN 0-262-22069-5, 2004.

[22] M. Huntbach, Meta-interpreters and Partial Evaluation in Parlog. *Formal Aspects of Computing* 1 (1) pp 193-211, 1989.