Specification and Implementation of a Multi-Agent Calculus based on Higher-Order Functions

Tony Clark
Department of Computing
University of Bradford, West Yorkshire, BD7 1DP, UK
a.n.clark@scm.brad.ac.uk

June 26, 1999

Abstract

Agents are autonomous system components that communicate using message passing. This paper presents a higher-order agent calculus and its implementation in the lazy functional programming language EBG. The calculus is given a semantics using a translation to the π-calculus that encodes higher-order functions and normal order evaluation.

1 Introduction

There is increasing demand for distributed software that is able to work collaboratively. The components of such a system are often referred to as *agents* [Jen98]. Each agent works autonomously but is able to communicate with other agents by passing messages containing data of arbitrary complexity.

An agent is similar to a conventional program in that most of its computation is sequential and can be described using existing models. Agents differ from conventional computation with respect to inter-agent communication mechanisms. Messages may be sent asynchronously and are buffered in a queue by the receiver until they are processed. An agent handles each message in turn, if there are no waiting messages the agent suspends computation until a message arrives. This is similar to the Actor model of computation [Agh86] [Agh91].

At any time an agent may create a new agent that will have a unique identity. The parent and child both know the identity of the child and can therefore communicate. Once the child is created, the computation in the child and parent continue concurrently. An agent dies (and may be garbage collected) when it chooses to ignore all further messages.

Given the overview of agent computation given above, it is proposed that conventional models may be used as the basis for computation within an agent and that an extension is added to the model to deal with agent creation and communication. This work aims to provide a notation for describing, executing and reasoning about agent computations. We take a λ-calculus with a normal order execution scheme [Han94] [Plo75] as our basis and extend it with primitive operators for agents.

An agent calculus is defined by extending λ-calculus terms with primitive agent operators. The syntax is defined and its semantics is given informally in section 2. A type theory [Car84] is developed identifying the terms of the calculus that correspond to agent programs in section 3. The calculus is given a rigorous semantics by developing a translation to the π-calculus [Mil93] in section 4. The calculus has been implemented as part of the lazy functional programming language EBG

[Cla99]. We give an overview of the implementation and an example of EBG agent programs in section 5.

2 An Agent Calculus

$$
\begin{aligned}
E &::= V \mid \lambda V.E \mid EE \mid CE^{*} \mid @EE \mid E \leftarrow E \mid \\
&\quad\; E \square E \mid \mathbf{mcase}\ \{A(;A)^{*}\}\ \mathbf{end} \mid \mathbf{skip} \\
A &::= P \rightarrow E \\
P &::= CP^{*} \mid V \mid K
\end{aligned}
$$

The terms for variables, functions and applications are conventional λ-terms. The term CE^{*} represents the application of a data constructor C to arguments. Data constructors start with capitals and variables start with lower case characters. All other terms are evaluated with respect to a current agent. $@MN$ causes the creation of a new agent with unique identity a. Both M and N are supplied with a and continue concurrently. The new agent behaviour is Ma and the current agent behaviour is Na. $M \leftarrow N$ causes message N to be sent to agent M. Execution continues at the current agent without waiting for a reply. The message is added to the target agent's message queue. $M \square N$ causes M to be performed before N; the value of M is ignored. **mcase** $\tilde{a}$ **end** causes the next message to be removed from the current agent's message queue. If the queue is empty then the agent blocks until a message is received. Each a_i consists of a pattern p_i and a term M_i. The message is matched against each of the patterns in turn. If it matches a pattern then any variables are bound and the corresponding term is evaluated. If no pattern matches then the message cannot be handled and the **mcase** term has no further effect. **skip** does nothing and produces the unit value.

Consider the case of a two-state switch. Each time the switch is pushed it changes state. The switch may be queried to find out its current state. The switch is defined in the calculus as follows:

```
switch state self =
  mcase
    Push -> switch (not state) self;
    Get agent ->
      agent <- state []
      switch state self
  end
```

Suppose that an agent refers to two existing agents a_1 and a_2 both of which are waiting to be sent a switch. The agent may create a new switch agent and supply it to both a_1 and a_2: `@(switch true) \s.  a1 <- s [] a2 <- s.`

3 Agent Types

Not all λ-terms are correctly formed agent programs. An agent is a function that transforms a stream of input messages to a stream of output messages. The basic λ-calculus does not impose an ordering on term reduction. An agent calculus must impose an ordering so that the programmer can control agent communication.

We use *continuation passing* [Plo75] to impose an ordering on the execution of agent programs. Agent types are:

$$
\begin{gathered}
\mu = \textit{messages}, \iota = \textit{agent identifiers}, [\tau] = \textit{sequence of } \tau \\
\diamond = \textit{agent state}, \kappa = [\mu] \rightarrow \diamond \rightarrow [\mu], \alpha = [\mu] \rightarrow \diamond \rightarrow \kappa \rightarrow [\mu] \\
\tau_1, \tau_2, \ldots = \textit{types}
\end{gathered}
$$

$$
\begin{array}{ccc}
V \vdash v : V(v) & \dfrac{V[v \mapsto \tau_1] \vdash M : \tau_2}{V \vdash \lambda v.M : \tau_1 \to \tau_2} & \dfrac{\begin{array}{c} V \vdash M : \tau_1 \to \tau_2 \\ V \vdash N : \tau_1 \end{array}}{V \vdash MN : \tau_2} \\[2ex]
\dfrac{\begin{array}{c} V \vdash M : \iota \to \alpha \\ V \vdash N : \iota \to \alpha \end{array}}{V \vdash @MN : \alpha} & \dfrac{\begin{array}{c} V \vdash M : \iota \\ V \vdash N : \mu \end{array}}{V \vdash M \leftarrow N : \alpha} & \dfrac{\begin{array}{c} V \vdash M : \alpha \\ V \vdash N : \alpha \end{array}}{V \vdash M \Box N : \alpha} \\[2ex]
V \vdash \mathbf{mcase\ end} : \alpha & \begin{array}{c} \dfrac{\begin{array}{c} V \vdash \mathbf{mcase}\ \tilde{a_1}\ \mathbf{end} : \alpha \\ V \vdash \mathbf{mcase}\ \tilde{a_2}\ \mathbf{end} : \alpha \end{array}}{V \vdash \mathbf{mcase}\ \tilde{a_1}\tilde{a_2}\ \mathbf{end} : \alpha} \\ V \vdash \mathbf{skip} : \alpha \end{array} & \dfrac{\begin{array}{l} V[v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n] \vdash M : \alpha \\ FV(p) = \{v_1, \ldots .v_n\} \end{array}}{V \vdash \mathbf{mcase}\ p \to M\ \mathbf{end} : \alpha}
\end{array}
$$

Figure 1: Type Theory for Agent Calculus

$$
\begin{array}{ll}
(1) & \llbracket a \rrbracket = (s,p).\bar{p}<a> \\
(2) & \llbracket v \rrbracket = (s,p).\bar{v}<s,p> \\
(3) & \llbracket \lambda v.M \rrbracket = (s,p).\nu x \ \ \bar{p}<x>.x(v).\llbracket M \rrbracket \\
(4) & \llbracket MN \rrbracket = (s,p).\nu q \ \ (\llbracket M \rrbracket(s,q) \mid q(v).\nu x \ \ \bar{v}<x,s,p>.!x\llbracket N \rrbracket) \\
(5) & \llbracket @MN \rrbracket = (s,p).(\nu io)(\llbracket Mi \rrbracket(o,\epsilon) \mid \llbracket Ni \rrbracket(s,p) \mid Q(i,o)) \\
(6) & \llbracket M \leftarrow N \rrbracket = (s,p).\nu q \ \ (\llbracket M \rrbracket(s,q) \mid q(a).\bar{p}<a>.\nu x \ \ (\bar{a}<x> \mid x\llbracket N \rrbracket)) \\
(7) & \llbracket M \Box N \rrbracket = (s,p).\nu c \ \ (\llbracket M \rrbracket(s,c) \mid c(\_).\llbracket N \rrbracket(s,p)) \\
(8) & \llbracket \mathbf{mcase}\ p_i \to M_i\ \mathbf{end} \rrbracket = (s,p).s(m).\nu q \ \ (\bar{m}<s,q> \mid \Sigma_{i=1,n} q(p_i).\llbracket M_i \rrbracket(s,p)) \\
(9) & \llbracket \mathbf{skip} \rrbracket = (s,p).\bar{p}<> \\
(10) & Q(i,o) = \nu q \ \ Q'(i,q) \mid \bar{q}<o> \\
(11) & Q'(i,c) = i(x).\nu q \ \ (c(o).\bar{o}<x>.\bar{q}<o> \mid Q'(i,q))
\end{array}
$$

Figure 2: Agent Semantics

μ is the type of messages sent between agents, κ is the type of agent continuations, α is the type of agent programs and $\Diamond$ is the type of agent states. An agent is supplied with a stream of input messages, a state and a continuation. The agent performs some commands, perhaps consuming and producing messages, and then proceeds by supplying the continuation with the rest of the input messages and the state.

All agent commands are parameterised with respect to a continuation. A linearity constraint prevents the continuation from being duplicated (in the implementation the continuation is hidden from the programmer behind the agent API), therefore the order of agent commands is fixed with respect to a single agent.

Figure 1 defines a type theory for the agent calculus. The theory defines a relation $V \vdash M : \tau$ holding between an environment, mapping variables to types, an agent calculus term M and a type τ. The intended meaning is that M has type τ when V associates free variables in M with their types.

4 Agent Semantics

The class of sequential programming languages is often given a semantics by a translation into a λ-calculus whose execution is described by a reduction relation between terms. Although the agent calculus contains sequential features (the λ-calculus sub-language used to describe calculations within an agent) the inclusion of concurrent features renders λ-reduction unusable as a semantic framework [San99].

The π-calculus [Mil93] is a formal system suitable for describing concurrent behaviour. Terms in the π-calculus represent processes that communicate using named channels. The agent calculus is higher-order since it includes all λ-terms as a sub-language. The basic π-calculus is not higher-order but there is a standard translation from λ-terms to π-terms that encodes higher-order features [San99].

$$
\begin{aligned}
&Q(i,o) \mid \bar{i}{<}v_1{>}.\bar{i}{<}v_2{>} \mid o(x_1).o(x_2) \\
&= Q'(i,q_1) \mid \bar{q_1}{<}o{>} \mid \bar{i}{<}v_1{>}.\bar{i}{<}v_2{>} \mid o(x_1).o(x_2) \\
&= i(x).(q_1(o).\bar{o}{<}x{>}.\bar{q_1}{<}o{>} \mid Q'(i,q_1)) \mid \bar{q_1}{<}o{>} \mid \bar{i}{<}v_1{>}.\bar{i}{<}v_2{>} \mid o(x_1).o(x_2) \\
&= q_1(o).\bar{o}{<}v_1{>}.\bar{q_1}{<}o{>} \mid Q'(i,q_1) \mid \bar{q_1}{<}o{>} \mid \bar{i}{<}v_2{>} \mid o(x_1).o(x_2) \\
&= \bar{o}{<}v_1{>}.\bar{q_1}{<}o{>} \mid Q'(i,q_1) \mid \bar{i}{<}v_1{>} \mid o(x_1).o(x_2) \\
&= \bar{q_1}{<}o{>} \mid i(x).(q_1(o).\bar{o}{<}x{>}.\bar{q_2}{<}o{>} \mid Q'(i,q_2)) \mid \bar{i}{<}v_2{>} \mid o(x_2) \\
&= \bar{q_1}{<}o{>} \mid q_1(o).\bar{o}{<}v_2{>}.\bar{q_2}{<}o{>} \mid Q'(i,q_2) \mid o(x_2) \\
&= \bar{o}{<}v_2{>}.\bar{q_2}{<}o{>} \mid Q'(i,q_2) \mid o(x_2) \\
&= \bar{q_2}{<}o{>} \mid Q'(i,q_2)
\end{aligned}
$$

Figure 3: Queue Behaviour

The agent calculus is given a semantics using a translation to the π-calculus. The translation is an extension of that which translates λ-terms with a *normal order* reduction relation. The translation is defined in figure 2. The overall effect is to translate each agent program into a process that awaits two names s and p. The process represents a single agent. Messages sent to the agent are received at the name s which may be thought of as identifying the agent's thread of control. The name p is used for co-ordination in an agent; the agent completes its current task and then delivers the result at p.

The translation of λ-terms encodes a normal order reduction strategy in the π-calculus. In (1), an agent a is supplied at p. In (2) a variable v is forced by supplying s and p. In (3) a λ-function is translated as a new channel x. When a value v is supplied at x the body M is performed. In (4) normal order application is encoded by evaluating M and producing the operator at q. The channel v is supplied with a new channel x that will perform the argument N when sent a message. In (5) an agent is created by generating a new message queue $Q(i,o)$ using an input channel i and an output channel o. The new agent, Mi continues concurrently with the current agent Ni. The channel ϵ is used as a message sink. In (6) the evaluation of the message N is delayed by creating a new channel x that performs N when sent a message. The term M is performed and produces an agent a at q. In (7), evaluation of M and N are ordered using a local channel c. In (8) the next message is received at s. If no message is available then the agent's execution will halt until a message is supplied by its buffer. A message is forced by supplying it with names s and q. The result is supplied at q. Each arm contains a pattern p_i and is encoded as an alternative for values received at q. When a value matches a pattern, the corresponding term M_i is performed. In (9), the null operation supplies the unit value to the continuation channel p.

Agent communication occurs through message queues. A message queue is created using (10) and (11). Each message queue $Q(i,o)$ contains two channels, the input i and the output o. Sequences of messages arrive at the input and are available in the same arrival order at the output. An example queue behaviour is shown in figure 3 in which two messages v_1 and v_2 are sent and then produced in the same order.

A rigorous semantics serves two purposes: an unambiguous way of defining what we mean by executing terms in the agent calculus and also a way of proving properties about agent programs. Suppose that we want to establish the following theorem:

Theorem 1 *Sequencing in agent programs is associative, i.e.* $M \Box (N \Box O) = (M \Box N) \Box O$ *for all terms* M, N *and* O.

Proof 1 *The translation* $[\![.]\!]$ *produces the following terms for each side of the equa-*

tion:

$$(s,p).\nu c\ \ ([\![M]\!](s,c)\mid c(\_).\nu c\ \ ([\![N]\!](s,c)\mid c(\_).[\![O]\!](s,p)))$$
$$(s,p).\nu c\ \ (\nu c\ \ ([\![M]\!](s,c)\mid c(\_).[\![N]\!](s,c))\mid c(\_).[\![O]\!](s,p))$$

Using standard properties of the π-calculus we may consistently rename locals and lift them to an outer scope. This produces terms of the following form:

$$P(c_1)\mid c_1.(Q(c_2)\mid c_2.R) \tag{1}$$

$$(P(c_1)\mid c_1.Q(c_2))\mid c_2.R \tag{2}$$

where $X(c)$ means perform an action using c as a shared communication channel and $c.Y$ means accept an input on the shared communication channel c then perform Y.

The translation $[\![.]\!]$ uses a shared communication channel p to co-ordinate execution. By inspection, p is linear since it is only used once in each equation in figure 2. Therefore, any term of the form $X(c)|c$ has a deterministic behaviour. The final part of the proof uses this result to show that terms 1 and 2 are behaviourally equivalent.

Consider term 1, the behaviour is P until a result is produced at c_1, then Q until a result is produced at c_2 then R. Term 2 produces the same behaviour. Since the terms are closed and each term has only one possible behaviour we conclude that they are behaviourally equivalent as required.

5 Agent Implementation

An agent is a function that processes a stream of messages. An agent's input stream is provided by its environment and the resulting output stream is processed by the environment. In addition to a stream of messages, the environment supplies data to be used as an *agent life-support system.* This section describes how the agent calculus is implemented as part of the lazy functional programming language EBG [Cla99].

Agents communicate by sending *messages.* A message may be *asynchronous* meaning that the source agent does not expect a return value, or may be *synchronous* meaning that the source agent waits for a return value. A message contains data and is *named* when the data is associated with a string (usually used for dispatching to a message handler in the target), otherwise it is *anonymous.* The EBG type `message` is:

```
type message =
  Message string $   ;;; Asynchronous, named.
| Message0 $         ;;; Asynchronous, anonymous.
| Call string int $  ;;; Synchronous, named.
| Call0 int $        ;;; Synchronous, anonymous.
| Return int $;      ;;; Return value.
```

Agent *identifiers* are used to refer to agents in message packets. An agent identifier is implemented as an integer. A *message packet* is a triple `(src,tgt,msg)` where `src` is the identifier of the source agent, `tgt` is the identifier of the target agent and `msg` is the message. The type `packets` of message packets is defined in EBG as follows:

```
type agentId = int;
type packet = (agentId,agentId,message);
type packets = list packet;
```

The types `agentId` and `packet` implement the agent calculus types ι and μ respectively. An *agent* `a` is a function that is applied to 6 arguments and returns a sequence of message packets:

```
a self in cont value coord os => out
```

where `self` is the agent's own identifier, `in` is a sequence of input message packets, `cont` is an agent continuation, `value` is the most recently received return value, `coord` is used for co-ordinating synchronous messages, `os` is the identifier of the operating system agent and `out` is the sequence of output message packets produced by `a`.

Typically, an agent `a` will consume some prefix `in'` of `in` leaving `in''` and producing packets `out'` such that `out = out' ++ out''`. The continuation `cont` is then supplied with the rest of the packets `in''` and produces `out''`. The continuation allows agents to be composed using a sequencing operator `then` (see below). The type `agent` is defined in EBG as follows:

```
type messageId = int;
type replace = agentId packets $ messageId agentId -> packets;
type agent = agentId packets replace $ messageId agentId -> packets;
```

The types `replace` and `agent` implement the agent calculus types κ and α respectively. In the calculus, the extra argument types in `agent` are bundled together into a single state type $\diamond$.

Agents are constructed using command primitives. The base layer of the agent model implements agent communication. The `comm` operator is supplied with a target agent identifier and a message. The message is delivered to the target. If the message is synchronous then the source agent will wait until the return value is received otherwise `comm` returns control to the agent via `cont`:

```
comm :: agentId message -> agent;
comm tgt msg = \self in cont value coord os.
  case msg of
    Message name data -> (self,tgt,msg):(cont self in value coord os);
    Message0 data     -> (self,tgt,msg):(cont self in value coord os);
    Return id data    -> (self,tgt,msg):(cont self in value coord os);
    Call name id data -> (self,tgt,msg):(wait id self in cont coord os []);
    Call0 id data     -> (self,tgt,msg):(wait id self in cont coord os [])
  end;
```

The `comm` operator uses `wait` to buffer input packets until the required return value is received. The operator is supplied with 7 values:

```
wait id self in cont coord os buff => out
```

where `id` is a message identifier and `buff` is a sequence of packets. An agent sends a synchronous message by producing a message `Call name id data`. The `id` component is a message identifier supplied to the target of the message. The target produces a return value by sending a message `Return id value`. The source agent uses the `id` value to match the return value with the original call.

During the call, the source agent is still active and may receive messages which are buffered by adding them to the sequence `buf`. There are many different possible strategies for handling call and return. The `wait` operator:

```
wait :: messageId agentId packets replace messageId agentId packets -> agent;
wait id self in cont coord os buff =
  case in of
    (src,_,Return id' data) : in' ->
      case id = id' of
        True -> cont self (buff ++ in') data coord os;
        False -> wait id self in' cont coord os (buff ++ [head in])
      end;
    else wait id self in' cont coord os (buff ++ [head in])
  end;
```

causes the source agent to continually buffer messages until the target agent returns a value. Once the value is received, the buffered messages are handled in the order that they were received by adding them back into the input stream.

The agent calculus term $M \leftarrow N$ is implemented as `comm M N`. The agent calculus term $@MN$ is implemented in EBG by sending the operating system a request for a new agent and then supplying the new agent identifier to M and N:

```
opSys \o (agent (self \s. M s)) $then result \a. N a
```

In addition to a stream of message packets, an agent is supplied with values that are used to manage messages and values. Each of the life-support values are accessed using the primitives `self`, `result`, `seqVal`, `incSeq` and `opSys`. They have similar definitions for example:

```
self :: (agentId -> agent) -> agent;
self fun = \self in cont value coord os.
  (fun self) self in cont value coord os;
```

The next message is consumed by the primitive `message`:

```
message :: (packet -> agent) -> agent;
message fun = \self in cont value coord os.
  case in of
    message : in' ->
      (fun message) self in' cont value coord os;
    else []
  end
```

The agent calculus term **mcase** $\tilde{p}_1 \rightarrow M_1; \ldots; p_n \rightarrow M_n$ **end** is implemented using the `message` primitive:

```
message \m. case m of p1 -> M1; ...; pn -> Mn end
```

Message passing is ultimately performed using the primitive `comm`. It is convenient to provide higher level primitives that distinguish between different types of messages. These primitives package up the information and then call `comm`:

```
send :: agentId string $ -> agent;
send target name data = comm target (Message name data);

call :: agentId string $ -> agent;
call target name data = seqVal \seq.
  incSeq $then
  comm target (Call name seq data);
```

`send` sends an asynchronous message; `call` sends a synchronous message. `send0` and `call0` use `Message0` and `Call0` message constructors but are otherwise the same as `send` and `call`. Note how synchronous message passing uses the `seqVal` and `incSeq` primitives to associate each message with a unique message identifier that will be used to recognise the return value if it is eventually received.

Agent control is provided using a command sequencing primitive `then` and an empty command `skip`:

```
skip :: agent;
skip self in cont value coord os = cont self in value coord os;

then :: agent agent -> agent;
then c1 c2 = \self in cont value coord os.
  c1 self in
    (\self in value coord os.
        c2 self in cont value coord os)
    value coord os;
```

The agent calculus term $M \square N$ is implemented as `M $then N`. Agents are created using the primitives `agent` and `javaAgent`. The `agent` primitive is used to create agents that are based on EBG functions. The function must be of type `agent`. The `javaAgent` primitive is used to create agents based on Java classes. The name of the class is supplied as an argument to `javaAgent`. In both cases, agents are created by sending the operating system agent a message called `new`. The message is synchronous and the return value will be the agent identifier of the newly created agent. Agent creation primitives are defined as follows:

```
agent :: agent -> agent;
agent behaviour = opSys \os. call os "new" behaviour;

javaAgent :: string -> agent;
javaAgent className = opSys \os. call os "new" className;
```

Consider a cell that contains an integer value. The cell may be sent an asynchronous named message `inc` causing the value to be incremented by 1. The cell may be sent a synchronous named message `get` in which case the cell replies with its current value. The `cell` behaviour is implemented as a function:

```
cell :: int -> agent;
cell n =
  mcase
    (_,_,Message0 "inc") ->
      cell (n + 1);
    (src,_,Call0 coord "get") ->
      (reply src coord n) $then
      cell n;
    else cell n
  end;
```

Notice how `cell` uses `n` as its internal state. On receiving a message, `cell` supplies a replacement behaviour by calling itself with a new value for the state variable. On receiving `get`, the reply should be sent to the source agent `src`; `coord` allows the source agent to associate the reply with the original call.

We wish to create a single cell which is shared between two agents. The first agent continually sends the cell `c`, supplied as an argument, requests to increment its contents:

```
setter :: agentId -> agent;
setter c = (send0 c "inc") $then setter c;
```

The `setter` agent has a constant state `c` whose value is the agent identifier of a cell. The second agent continually sends the cell requests for its value:

```
getter :: agentId agentId -> agent;
getter c os =
  (call0 c "get") $then
  result \n.
    (send os "println" n) $then
    getter c os;
```

Since `get` is a synchronous message, the `getter` agent waits until the cell `c` replies with a value. The return value is accessible using the `result` primitive. The `getter` agent has two state components, `c` and `os`, that never change.

```
main =
  @(K (cell 0)) \c.
   opSys \os.
     (agent (setter c)) $then
     agent (getter c os)
```

Agents are created by supplying the `agent` primitive (called on the left via `@`) with a value of type `agent`. Since `agent` is implemented using `call`, `main` waits for the cell agent `c` then supplies it `setter` and `getter` agents.

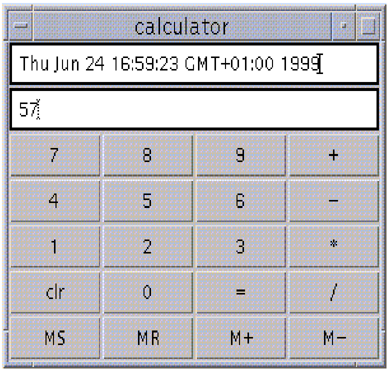
EBG uses agent primitives to provide support for state based and event driven multi-processing within a lazy functional programming language. EBG is implemented using a compiler that produces Java virtual machine byte codes. Agents provide an ideal mechanism for mixed EBG and Java programming. The following

```
op1 mem n =
  mcase
    (src,_,Message "enter" m) ->
      (send src "display" ((n*10)+m)) $then
      op1 mem ((n * 10) + m);
    (src,_,Message0 "clr") ->
      (send src "display" 0) $then
      op1 mem 0;
    (src,_,Message0 "MS") -> op1 n n;
    (src,_,Message "operator" op) ->
      op2 mem n op 0;
    else op1 mem n
  end;

calc n "+" m = n + m;   ;;; and for -,*,/

dateLoop jcalc =
  opSys \os.
    (call os "date" []) $then
    result \d.
      (send jcalc "date" d) $then
      dateLoop jcalc;
```

```
op2 mem n op m =
  mcase
    (src,_,Message "enter" m') ->
      (send src "display" ((m*10)+m')) $then
      op2 mem n op ((m * 10) + m');
    (src,_,Message0 "clr") ->
      (send src "display" 0) $then op1 mem 0;
    (src,_,Message0 "=") ->
      let result = calc n op m
      in (send src "display" result) $then
         op1 mem result;
    else op2 mem n op m
  end;

main =
  (agent (op1 0 0)) $then
  (result \calc.
    (javaAgent "CalcInterface") $then
    result \jcalc.
      (agent (dateLoop jcalc)) $then
      send jcalc "register" calc)
```

Figure 4: An EBG Calculuator Agent

is simple calculator that is implemented in both Java and EBG:



The GUI is implemented as a Java agent; the calculations and memory are implemented as an EBG agent. Events are generated by AWT buttons causing messages to be sent from the Java agent to the EBG agent. Part of the EBG agent is shown in figure 4. The `op1` behaviour handles operations for the left hand operator and `op2` for the right. `n` is the current number and `m` is the memory.

Clicking a number sends an `enter` message from the Java `CalcInterface` agent to the EBG agent. The EBG agent sends the Java agent `display` and `date` messages to update the numeric and date display regions.

6 Conclusion

This paper has defined an agent calculus and given it a semantics using a translation to the π-calculus that encodes higher-order functions and normal order evaluation. The calculus has been implemented in the lazy functional programming language EBG as a collection of primitives. The primitives have been shown to support state-based, event driven, mixed paradigm programming in terms of the implementation of a small calculator program. Streams have been used as the basis of concurrency, reactivity and state in functional programs, for example [Car98], [O'D85], [Tho90] and [Wad90]. There are a number of functional programming languages such as Concurrent ML and Concurrent Haskell that support multi-processing. This work is novel because it is based on a simple agent calculus with a precise semantics and is implemented in EBG and therefore supports the features of the Java VM [Ven98].

References

[Agh86] Agha, G.: Actors: *A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh91] Agha, G.: The Structure and Semantics of Actor Languages. In proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, LNCS 489, Springer-Verlag, 1991.

[Car84] Cardelli L. (1984) Basic Polymorphic Type Checking. *Science of Computer Programming,* **8**(2), 147 – 72.

[Car98] Carlsson M. & Hallgren T. (1998): Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces. PhD Thesis, Department of Computing Science, Chalmers University of Technology.

[Cla99] Clark, A. N. (1999): EBG: A Lazy Functional programming Language Implemented on the Java Virtual Machine. Technical Report submitted to the Computer Journal.

[Jen98] Jennings, N. R., Sycara, K & Wooldridge M. (1998): A Roadmap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems, 1, 7 – 38.

[Han94] Hankin C. (1994) *Lambda Calculi a Guide for Computer Scientists.* Clarendon Press, Oxford University Press.

[Mil93] Milner R. (1993): The Polyadic π-Calculus: A Tutorial. In F. L. Hamer, W. Brauer and H. Schwichtenberg, editors, Logic and Algebra of Specification. Springer-Verlag, 1993.

[O'D85] O'Donnell, J. T. (1985): Dialogues: A Basis for Constructing Programming Environments. SIGPLAN Notices 20(7):19 – 27.

[Plo75] Plotkin G. (1975) Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science.* **1**, pp 125 – 159.

[San99] Sangiorgi D. (1999): Interpreting functions as π-calculus processes: a tutorial. INRIA Technical Report RR-3470.

[Tho90] Thomson, S. (1990): Interactive Functional Programming. In Research Topics in Functional Programming, ed. Turner, D. A. Addison-Wesley.

[Ven98] Venners B. (1998) *Inside the Java Virtual Machine.* McGraw-Hill.

[Wad90] Wadler, P. (1990): Comprehending Monads. In Proc. 19th Symposium on Lisp and Functional Programming, Nice, ACM.