Implementation of Lazy Agents in the Functional Language EBG

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

July 15, 1999

Abstract

EBG is a lazy functional programming language that compiles to the Java Virtual Machine Language. The aims of EBG are to provide the benefits of both FP and Java. This paper describes the design and implementation of *agents* in EBG that provides an interface to the underlying multi-processing facilities of Java.

1 Introduction

EBG [Cla99a] is a higher-order lazy functional programming language that compiles to the Java Virtual Machine [Ven98]. EBG aims to provide all of the advantages of FP including pattern matching, first class functions and automatic type checking [Fie89], in addition to the advantages of Java [Arn98] including portability, multi-processing, networking and graphical user interfaces.

This paper describes the design and implementation of *agents* in EBG. Agents provide an EBG-level interface to the multi-processing facilities of Java and are a step on the path to a longer term goal of providing programming facilities for functional multi-agent systems [Jen98]. Agents are based on both the Actor model of computation [Agh86] [Agh91] and models processes as stream consumer-producer functions [Tho90] [O'D85].

An agent is a function that processes a stream of input messages and produces a stream of output messages. Agents execute concurrently and communicate by passing messages. A message is sent from a source agent to a target agent. A source agent sends a message to its output stream and then continues without waiting for a reply. The message is transferred to the target agent's input stream.

Messages are consumed at an agent by processing the elements of its input stream in the order that they arrive. When an agent requests an input the next

expressions $M, N, \ldots$	$E ::=$	V	variables v, w
		$\mid \lambda V.E$	functions $\lambda v.M$
		$\mid EE$	applications MN
		$\mid KE^*$	structures $k\tilde{M}$
		$\mid$ **case** E **of** A^* **end**	selection $(M, \tilde{a})$
		$\mid C$	agent command
commands	$C ::=$	$\mid\ \leftarrow E$	send message
		$\mid$ **skip**	no operation
		$\mid E; E$	sequence
		$\mid$ **mcase** A^* **end**	message dispatch
case arms a	$A ::=$	$P \rightarrow E$	case arm $p \rightarrow M$
patterns p	$P ::=$	V	variable pattern v
		$\mid KV^*$	structure pattern $k\tilde{v}$

Figure 1: Agent Calculus

available message is removed. If no messages are available then the agent blocks until the next message is received.

This paper is structured as follows. Section 2 defines an agent calculus that represents the essential features of the agent model of computation. The semantics of the calculus is defined as an equivalence relation on agent terms and as such provides scope for reasoning about agent properties without reference to implementation details. Section 3 refines the semantics so that it is *computational* by adding control features in the form of a state transition machine. Section 4 describes how the calculus is implemented in EBG as an agent API and section 5 describes how the mechanisms are implemented in Java as part of the EBG run-time system.

2 An Agent Calculus

An agent is implemented as a program written in an *agent calculus* (see figure 1) which is λ-calculus extended with structures, pattern matching and agent commands. The commands allow messages to be sent ($\leftarrow M$) and received (**mcase**...**end**). Agent messages are sequenced using the operator ; whose left and right identity is the empty command **skip**. The calculus has no special syntax for creating new agents, this is achieved by sending messages to a distinguished *operating system* agent.

2.1 A Theory of Agents

The standard notion of term equivalence ($=$) [Han94] is extended for the agent calculus. The definition has two parts: equivalence between expressions and equivalence between systems of agents. Equivalence between expressions is de-

$$(\lambda v.M)N = M[v := N] \quad (1)$$

$$\frac{M = M' \; N = N'}{MN = M'N'} \quad (2)$$

$$\frac{M = N}{\lambda v.M = \lambda v.N} \quad (3)$$

$$\frac{M_1 = M_1' \ldots M_n = M_n'}{kM_1 \ldots M_n = kM_1' \ldots M_n'} \quad (4)$$

$$\mathbf{skip}; M = M = M; \mathbf{skip} \quad (5)$$

$$M; (N; O) = (M; N); O \quad (6)$$

$$\frac{M = M' \; N = N'}{M; N = M'; N'} \quad (7)$$

$$\frac{M = M'}{\leftarrow M = \leftarrow M'} \quad (8)$$

$$\frac{M = M'}{\mathbf{case}\ M\ \mathbf{of}\ \tilde{a}\ \mathbf{end} = \mathbf{case}\ M'\ \mathbf{of}\ \tilde{a}\ \mathbf{end}} \quad (9)$$

$$\mathbf{case}\ k\tilde{M}\ \mathbf{of}\ \tilde{a}_1\ k\tilde{v} \to N\ \tilde{a}_2\ \mathbf{end} = N[\tilde{v} := \tilde{M}] \quad (10)$$

Figure 2: Equivalence of Agent Terms

$$\frac{M = M'}{\{(i, m, M)\} = \{(i, m, M')\}} \quad (11)$$

$$\frac{\Sigma_1 = \Sigma_1' \; \Sigma_2 = \Sigma_2'}{\Sigma_1 \cup \Sigma_2 = \Sigma_1' \cup \Sigma_2'} \quad (12)$$

$$\{(i, k\tilde{M} : m, \mathbf{mcase}\ \tilde{a}_1\ k\tilde{v} \to N\ \tilde{a}_2\ \mathbf{end})\} = \{(i, m, N[\tilde{v} := \tilde{M}])\} \quad (13)$$

$$\{(i_1, m_1, (\leftarrow ki_1 i_2 M); N), (i_2, m_2, O)\} = \{(i_1, m_1, N), (i_2, m_2 +\!\!+ [ki_1 i_2 M], O)\} \quad (14)$$

$$\{(i, k\tilde{M} : m, \mathbf{mcase}\ \tilde{a}\ \mathbf{else}\ N\ \mathbf{end})\} = \{(i, k\tilde{M} : m, N)\} \text{ no match} \quad (15)$$

$$\{(i, m, \mathbf{mcase}\ \tilde{a}\ \mathbf{end})\} = \{(i, m, \mathbf{skip})\} \text{ no match} \quad (16)$$

Figure 3: System Semantics

fined by the theory given in figure 2 in addition to being reflexive, symmetric and transitive. Note that expression equivalence does not state anything about agent command equivalence except where the equivalence involves that of sub-expressions.

Agent systems are sets of agents. An agent (i, m, M) consists of a unique identifier i, a message queue m and an agent command M. System equivalence is given by the theory in figure 3 (in addition to being reflexive, symmetric and transitive) and is defined in terms of expression equivalence (rule 11). Message passing transfers a message from one agent to another (14). The message is added to the target agent's queue and is subsequently processed on demand. When processed, a message may match a pattern (13) or fail. When it fails, if there is a default arm (15) then the agent continues otherwise it terminates (16).

The basic calculus can be extended with built-in agents and messages. These are similar to the δ-rule extensions to a basic λ-theory. A distinguished *operating system* agent o implements an interface between the calculus and its environment. For example, dynamic agent creation is performed by:

$$\{(i, m, (\leftarrow io(new\ M)); N)\} = \{(i, m +\!\!+ [(o, i, j)], N), (j, [], M)\}$$

Term equivalences can be used to prove both agent and system properties. A typical example is to establish that two agents are (or are not) equivalent. When establishing agent equivalences we must ensure that all possible system cases are covered. Conversely, equivalence is denied by showing that there exists a system in which the agents behave differently.

Theorem 1 *Let* $A_1 = \textbf{mcase}\ p_1 \to M\ p_2 \to N\ \textbf{end}$ *and let* $A_2 = \textbf{mcase}\ p_1 \to M\ \textbf{else mcase}\ p_2 \to N\ \textbf{end}$*, then* A_1 *and* A_2 *are equivalent agents.*

Proof 1 *We must establish that* $\Sigma \cup \{(i, m, A_1)\} = \Sigma \cup \{(i, m, A_2)\}$ *for any system* Σ*, identifier* i *and message queue* m*. We proceed by cases with respect to the behaviour of the system. Suppose that* $m = []$ *and that no message is ever produced by* Σ *for agent* i*, then* A_1 *and* A_2 *have equivalent behaviours. Now suppose that* $m = []$ *and* Σ *produces a message, or that* m *is initially non-empty. If the message matches* p_1 *then both* A_1 *and* A_2 *become* M *by rule 13. Alternatively, if the message matches* p_2 *then both* A_1 *and* A_2 *become* N *by rule 13 and rules 13 and 15 respectively. Finally, if the message fails to match* p_1 *and* p_2 *then both* A_1 *and* A_2 *become* **skip** *by rule 16. Therefore the agents are behaviourally equivalent in all possible systems as required.*

Theorem 2 *Let* $A_1 = (\leftarrow M); \textbf{mcase}\ p \to (\leftarrow N)\ \textbf{end}$ *and let* $A_2 = \textbf{mcase}\ p \to (\leftarrow M); (\leftarrow N)\ \textbf{end}$ *then* A_1 *and* A_2 *are equivalent agents.*

Proof 2 *The theorem is false. To show that it is false we construct a system in which the two agents exhibit different behaviours. Let* $\{(i, [], A)\}$ *be a system and let* M *be a message from agent* i *to itself that matches pattern* p*. When* $A = A_1$ *the system produces an output message* N *and becomes* **skip**. *When* $A = A_2$ *the system exhibits no behaviour. Therefore, the agents exhibit different behaviours with respect to the same system.*

2.2 CPS for Agents

Agent commands may be implemented by a translation to basic λ-terms using a continuation passing style (CPS) [Plo75], see figure 4. Each agent command is a function expecting an input stream m and a continuation k. The command can consume elements of the input stream. When it is complete, the command passes the rest of the input stream to the continuation. The initial continuation is $\lambda m.[]$ which, when invoked, causes the agent to terminate. The **skip** command simply invokes the continuation (17). Messages are sent by adding them to the head of the output stream (18). Commands are sequenced by supplying the first M with a continuation to perform the second N with respect to the rest

$$[\![\mathbf{skip}]\!] = \lambda mk.km \tag{17}$$

$$[\![\leftarrow N]\!] = \lambda mk.N : (km) \tag{18}$$

$$[\![M;N]\!] = \lambda mk.Mm\lambda m.Nkm \tag{19}$$

$$\begin{array}{l} [\![\mathbf{mcase}\ \tilde{a}\ \mathbf{end}]\!] = \lambda mk. \\ \quad \mathbf{case}\ m\ \mathbf{of} \\ \quad\quad x : m' \rightarrow \\ \quad\quad\quad (\mathbf{case}\ x\ \mathbf{of}\ \tilde{a}\ \mathbf{end})km' \\ \quad\quad \mathbf{else}\ <> \\ \quad \mathbf{end} \end{array} \tag{20}$$

Figure 4: Translation to CPS

of the input stream (19). The next message is found by matching it against a collection of message patterns (20); if there are no input messages then the agent blocks by producing the special value $<>$. An agent is supplied with its complete input stream; for an agent to produce $<>$ there must be no messages either pending or sent to the agent in the future. The translation can be used to validate rules 5 and 6.

Theorem 3 $\mathbf{skip}; M = M = M; \mathbf{skip}$

Proof 3
$$\begin{aligned} \mathbf{skip}; M &= \lambda mk.\ \mathbf{skip}\ m\ \lambda m.Mmk \\ &= \lambda mk.Mmk = M \\ &= \lambda mk.Mm\lambda m.km \\ &= \lambda mk.Mm\lambda m.\ \mathbf{skip}\ m\ k \\ &= M; \mathbf{skip} \end{aligned}$$

Theorem 4 $M; (N; O) = (M;)N; O$

Proof 4
$$\begin{aligned} M; (N; O) &= \lambda mk.Mm\lambda m.(N; O)mk \\ &= \lambda mk.Mm\lambda m.Nm\lambda m.Omk \\ &= \lambda mk.(M; N)m\lambda m.Omk \\ &= (M; N); O \end{aligned}$$

2.3 Agent Types

An agent is an expression whose type is an agent command. The type of agent commands is $\alpha = [\mu] \rightarrow ([\mu] \rightarrow [\mu]) \rightarrow [\mu]$, the type of messages is μ and the type of agent identifiers is ι. A type theory associates each expression M with a type τ when $A \vdash M : \tau$ such that A associates free variables of M with types. The theory is standard [Car84] except for the agent commands:

$$\frac{A \vdash M : \mu}{A \vdash \leftarrow M : \alpha} \quad (21) \qquad \frac{A \vdash M : \alpha \; A \vdash N : \alpha}{A \vdash M; N : \alpha} \quad (22) \qquad A \vdash \textbf{skip} : \alpha \quad (23)$$

$$\frac{A[v_1 \mapsto \iota, v_2 \mapsto \iota, v_3 \mapsto \tau] \vdash M : \alpha}{A \vdash \textbf{mcase}\ \tilde{a}_1\ kv_1v_2v_3 \to M\ \tilde{a}_2\ \textbf{end} : \alpha} \quad (24)$$

3 Agent Computations

The agent theory is not directed and therefore does not indicate how agent calculations will take place in EBG. Intra-agent calculations will be deterministic given complete information about the agent's input stream. It will not be possible to impose a deterministic order on inter-agent calculations since we will abstract away from the details of the message delivery service. We distinguish between system execution (underspecified with respect to execution ordering) and agent execution (totally specified with respect to execution ordering). Agent execution must deal with blocking on input streams and forcing lazily generated output streams. This section refines the agent theory using a state transition machine semantics. The transition machine has been implemented as a functional program in EBG.

3.1 System Execution

An agent is represented as an SECD machine [Lan64] extended with components for the agent's unique identifier and message queue. A system is a set of agents and system behaviour is defined as a relation between sets of agent machine states.

An agent is either *active* or *terminated.* A terminated agent is one which has irretrievably ceased to process messages and serves as a sink for all messages it receives. An agent state is (i, m, γ) where i is a unique agent identifier, m is the agent's message queue and γ is an SECD state. An SECD state is either (s, e, c, d) or $()$. A terminated agent is $(i, m, ())$ and an active agent is $(i, m, (s, e, c, d))$ (written (i, m, s, e, c, d)).

Let Σ be a set, or *system,* of agent states. System execution is defined by a pair of relations, $\Rightarrow$ and $\longrightarrow$, such that $\Sigma \Rightarrow \Sigma'$ is system execution step and $\sigma \longrightarrow \sigma'$ is an agent execution step.

System execution steps are defined in figure 5. System execution consists of component agent execution (25). Each of the component agents may execute concurrently with all other agents (26). Time is relative to an agent, *i.e.* component agents do not all execute in lock step (27).

Axiom 28 delivers messages from one agent to another. A source agent (i_1) sends a message x by performing the machine instruction $\leftarrow x$. The target agent identifier (i_2) is on the source agent's stack. A message (i_1, i_2, x) is delivered by removing it from the source agent and adding it to the end of the target agent's message queue.

$$\frac{\sigma \longrightarrow \sigma'}{\Sigma \cup \{\sigma\} \Rightarrow \Sigma \cup \{\sigma'\}} \quad (25) \qquad \frac{\begin{array}{c}\Sigma_1 \Rightarrow \Sigma_1' \\ \Sigma_2 \Rightarrow \Sigma_2'\end{array}}{\Sigma_1 \cup \Sigma_2 \Rightarrow \Sigma_1' \cup \Sigma_2'} \quad (26) \qquad \Sigma \Rightarrow \Sigma \quad (27)$$

$$\begin{array}{l}\Sigma \cup \{(i_1, m_1, (i_2 : s, e, \leftarrow x : c, d)), (i_2, m_2, \gamma)\} \Rightarrow \\ \quad \Sigma \cup \{(i_1, m_1, (s, e, c, d)), (i_2, m_2 +\!\!+ [(i_1, i_2, x)], \gamma)\}\end{array} \tag{28}$$

Figure 5: System Execution

$$(i, m, ()) \longrightarrow (i, m, ()) \tag{29}$$

$$(i, m, s, e, v : c, d) \longrightarrow (i, m, [], e', [M], (s, e, c, d)) \text{ when } e(v) = (e', M) \tag{30}$$

$$(i, m, s, e, (\lambda v M) : c, d) \longrightarrow (i, m, (v, e, M) : s, e, c, d) \tag{31}$$

$$(i, m, s, e, (MN) : c, d) \longrightarrow (i, m, s, e, M : (N) : @ : c, d) \tag{32}$$

$$(i, m, s, e, (M) : c, d) \longrightarrow (i, m, (e, M) : s, e, c, d) \tag{33}$$

$$(i, m, x : (v, e', M) : s, e, @ : c, d) \longrightarrow (i, m, [], e'[v \mapsto x], [M], (s, e, c, d)) \tag{34}$$

$$(i, m, s, e, (M, \tilde{a}) : c, d) \longrightarrow (i, m, s, e, M : \tilde{a} : c, d) \tag{35}$$

$$(i, m, k\tilde{x} : s, e, (k\tilde{v} \rightarrow M : \tilde{a}) : c, d) \longrightarrow (i, m, [], e[\tilde{v} \mapsto \tilde{x}], [M], (s, e, c, d)) \tag{36}$$

$$(i, m, k\tilde{x} : s, e, (k'\tilde{v} \rightarrow M : \tilde{a}) : x, d) \longrightarrow (i, m, k\tilde{x} : s, e, \tilde{a} : c, d) \ k \neq k' \tag{37}$$

$$(i, m, s, e, k\tilde{M} : c, d) \longrightarrow (i, m, k(e, \tilde{M}) : s, e, c, d) \tag{38}$$

$$(i, m, x : \_, \_, \_, (s, e, c, d)) \longrightarrow (i, m, x : s, e, c, d) \tag{39}$$

Figure 6: Agent Execution

3.2 Agent Execution

Expressions in the agent calculus denote a closure (v, e, M), a structure $k\tilde{x}$ or an input message stream. A closure is created when a λ-function $\lambda v.M$ is evaluated and it captures the current machine environment e. A structure consists of a constructor k and a sequence of thunks $\tilde{x}$. A thunk (e, M) associates an expression M with an environment e containing bindings for all the free variables in M. An environment e is a partial function from variables to thunks and is extended with a binding between v and x producing $e[v \mapsto x]$ in the usual way. The term $k(e, \tilde{M})$ denotes the structure $k(e, M_1)(e, M_2) \ldots (e, M_n)$. The term $e[\tilde{v} \mapsto \tilde{x}])$ denotes the environment $e[v_1 \mapsto x_1, \ldots, v_n \mapsto x_n]$.

Agent execution is defined by the transition function given in figure 6. Agent states use *machine instructions*, they are: (M) to delay the evaluation of M; @ to apply an operator to an operand; and, $\tilde{a}$ to try each case arm in turn. A terminated agent (29) cannot perform any computation. The agent calculus

$$(i, m, (e', M) : hd : s, e, @ : c, d) \longrightarrow (i, m, [], e', [M], (hd : s, e, @ : c, d)) \quad (40)$$

$$(i, m, (e', M) : tl : s, e, @ : c, d) \longrightarrow (i, m, [], e', [M], (tl : s, e, @ : c, d)) \quad (41)$$

$$(i, m, (x : \_) : hd : s, e, @ : c, d) \longrightarrow (i, m, x : s, e, c, d) \quad (42)$$

$$(i, m, (\_ : x) : tl : s, e, @ : c, d) \longrightarrow (i, m, x : s, e, c, d) \quad (43)$$

$$(i, m_1 +\!\!+ [x] +\!\!+ m_2, \$n : hd : s, e, @ : c, d) \longrightarrow \left\{ \begin{array}{ll} (i, m_1 +\!\!+ [x] +\!\!+ m_2, x : s, e, c, d) & \text{when } \#m_1 = n \\ (i, m_1 +\!\!+ [x] +\!\!+ m_2, \$n : hd : s, e, @ : c, d) & \text{when } \#m_1 + \#m_2 < n \end{array} \right. \quad (44)$$

$$(i, m, \$n : tl : s, e, @ : c, d) \longrightarrow (i, m, \$n + 1 : s, e, c, d) \quad (45)$$

Figure 7: Handling Message Streams

uses a normal order execution scheme; therefore variables are bound to delayed expressions (thunks) in the current environment and must be *forced* (30) when they are required. Function expressions produce closures (31). Application evaluates the operator and delays the operand (32 and 33); when the operator is applied a fresh context is created (34) the result is returned to the original context (39). Pattern driven selection amongst alternatives is driven by the constructor (35 36 and 37). Structure creation delays the evaluation of the component expressions (38).

3.3 Agent Streams

An agent is a function that processes streams of messages. The streams are generated lazily and messages are added to a target agent's input stream as they are produced by the source agents. Output streams are built using the usual list constructor $\_ : \_$.

Input streams are represented using a special value $\$n$ where n is an integer; the meaning of the value is given in terms of the message queue component of an agent state. Taking the head of an input stream produces the message at the head of m after dropping n messages. Taking the tail of an input stream produces a value $\$n + 1$.

Message stream manipulation is performed using the list accessor operators *hd* and *tl* as defined in figure 7. The operators are strict and must force their arguments to produce cons-pairs or streams (40 and 41). If the accessors are applied to ordinary cons-pairs (42 and 43) then they produce the appropriate component. If *hd* is applied to a message stream (44) then, if there is a message currently available it is returned, otherwise the agent cannot satisfy the application and it blocks. If *tl* is applied to a message stream (45) then the result is a new message stream with an increased message index.

An agent sends messages by producing a sequence of pairs (i, x) where i

$$(i, m, [[]], e, [], ()) \longrightarrow (i, m, ()) \tag{46}$$

$$(i, m, [(e', M) : x], e, [], ()) \longrightarrow (i, m, [(e', M), x], e, [], ()) \tag{47}$$

$$(i, m, (e, M) : s, \_, [], ()) \longrightarrow (i, m, s, e, [M], ()) \tag{48}$$

$$(i, m, (x_1, x_2) : s, e, [], ()) \longrightarrow (i, m, x_1 : s, e, [\leftarrow x_2], ()) \tag{49}$$

Figure 8: Sending Messages

is the identifier of the target agent and x is an arbitrary data value. Agents execute lazily and therefore produce a sequence $x_1 : x_2$ where x_1, x_2 are thunks. Figure 8 shows the transition machinery necessary for sending messages. All of the transitions refer to completed agent computations; therefore the control is empty [] and the stream of output messages is on the stack (47). The head of the output message stream is forced (47 and 48). The target of the head message is forced (49) leaving the target on the stack ready for a system transition (28). When an agent ceases to produce messages it is terminated (46).

4 Agent Primitives

EBG provides an agent API that implements the agent calculus using a CPS encoding. The novel agent language mechanisms involve the underlying implementation of EBG and the API operators simply manage input streams and agent continuations. This section describes the implementation of the API operators and shows how synchronous message passing is layered on top of the basic asynchronous mechanism.

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* (of type μ in section 2.3) 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` ($[\mu]$) of message packets is defined in EBG as follows:

```
type agentId = int;
```

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

Agents are extended in the API with extra arguments. The 6 agent arguments, in order, are: the agent's own identifier; an input stream; a continuation; the most recent result of a synchronous message; a value used to coordinate call and return; and, the agent identifier of the operating system agent. The type `agent` (α in section 2.3) 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 agent command $\leftarrow$ is implemented using the API operator `comm`. In the case of asynchronous messages the operator returns the message. Synchronous messages use the `wait` operator:

```
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, the first being a message identifier `id` and the last being a message buffer `buf`.

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.

In addition to a stream of message packets, an agent is supplied with values that are used to manage messages and values. Each of these 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 operator `message` such that the agent calculus **mcase** $\tilde{a}$ **end** is implemented as `message \m. case m of` $\tilde{a}$ `end`:

```
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
```

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 when it is received.

Agent control is provided using a command sequencing primitive `then` (; in the calculus) 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;
```

Agents are created using the command `agent` that is applied to an EBG function of type `agent`. Agents are created by sending the operating system agent a `new` message. The message is synchronous and the return value will be the agent identifier of the newly created agent:

```
abstract class Thunk extends Value
{
  private Value cache = null;

  public abstract Value value();

  public Value force()
  {
    if(cache == null) {
      Value value = value();
      cache = value.force();
      return cache;
    } else return cache;
  }
}
```

```
class MessageStream extends Thunk
{
  private Queue queue;

  public MessageStream(Queue queue)
  { this.queue = queue; }

  public Value value()
  {
    while(queue.isEmpty()) {
      Thread.yield();
    }
    Value m = (Value)queue.next();
    queue.drop();
    MessageStream ms = new MessageStream(queue);
    return cons(m,ms);
  }
}
```

Figure 9: Implementation of Message Stream

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

5 Java Implementation

The novel agent execution mechanisms are implemented by the underlying EBG run-time system. Two new types of EBG value are required: message streams and agents. This section describes how these values are implemented based on EBG thunks and Java threads.

The EBG compiler delays function arguments by translating them to (instantiations of) sub-classes of `Thunk`. A thunk has a method `value` that delivers the value of the EBG expression when it is called. EBG evaluates *lazily*, each thunk has a cache that holds the value after it has been forced the first time. `Thunk` is defined in figure 9.

The input stream of an agent is a delayed value that is produced gradually as system computation proceeds. The act of forcing an input stream causes the next message to be requested from an agent's queue. If the queue is currently empty then the request is blocked until a message is received. Agent blocking does not affect system computation since each agent is implemented as a separate Java thread. Input message streams are based on `Thunk` in figure 9; the cache guarantees referential transparency.

Figure 10 shows how agents are implemented as part of the EBG run-time system. The system distinguishes between three types of agent: functional agents that are based on EBG closures; operating system agents that provide an interface to the system environment; and, Java agents (not shown) that provide a transparent interface between EBG and Java programs.

All agents are based on the abstract class `Agent`. Each agent has a unique `ident` and a message `queue`. The lookup table `agents` is global and associates

```
abstract class Agent extends Thread
{
 protected AgentId ident;

 static Hashtable agents;

 protected Queue queue;

 public Agent(AgentId ident)
 { this.ident = ident; }

 public void send(Value m)
 {
  AgentId tgt = target(m);
  Agent agent;
  agent = (Agent)agents.get(tgt);
  agent.receive(m);
 }

 public void receive(Value m)
 { queue.add(m); }

 AgentId newFunAgent(Closure f)
 {
  AgentId i = new AgentId();
  FunAgent a = new FunAgent(i,f);
  agents.put(i,a);
  a.start();
  return i;
 }
}


class FunAgent extends Agent
{
 Closure fun;

 public FunAgent(AgentId i,Closure f)
 {
  super(i);
  this.fun = f;
 }

 public void run()
 {
  MessageStream in;
  in = new MessageStream(queue);
  Value out = fun.apply(in);

  while(isCons(out)) {
   Value m = head(out);
   send(m);
   out = tail(out);
   yield();
  }
 }
}
```

```
class OperatingSystem extends Agent
{
 public OperatingSystem(AgentId i)
 { super(i); }

 public void run()
 {
  while(!queue.isEmpty()) {
   handlePackage();
   yield();
  }
  System.exit(0);
 }

 void handlePackage()
 {
  Value m = (Value)queue.next();
  queue.drop();
  AgentId src = messageSource(m);
  Value data = messageData(m);
  if(isMess(data))
   async(src,data);
  else sync(src,data);
 }

 void sync(AgentId src,Value call)
 {
  String name = callName(call);
  Value seq = callSeq(call);
  Value data = callData(call);
  Value res = async(src,name,data);
  Value ret = ret(seq,res);
  send(message(ident,src,ret));
 }

 Value async(AgentId src,Value m)
 {
  String name = messName(m);
  Value data = messData(m);
  return async(src,name,data);
 }

 Value async(AgentId src,String n,Value v)
 {
  if(n.equals("print"))
   return handlePrint(v);
  else if(n.equals("new"))
   return newFunAgent((Closure)v.force());
  else throw new Error("message? " + n);
 }

 Value handlePrint(Value value)
 {
  value.print(stdout);
  return value;
 }
}
```

Figure 10: Implementation of Agent

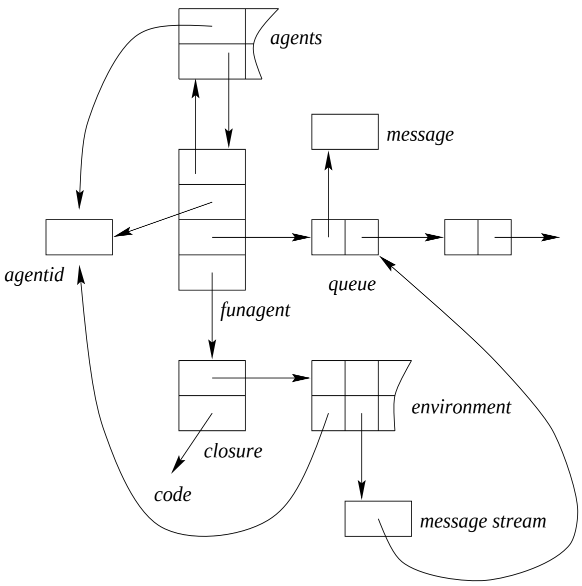

Figure 11: Agent Structure

identifiers and agents. An agent sends and receives messages using `send` and `receive` respectively.

The EBG compiler translates λ-functions to (instantiations of) sub-classes of `Closure`. Each sub-class of `Closure` must define a method `apply` that delivers the result of applying the λ-function when it is supplied with an argument. A `FunAgent` is based on an EBG `closure`. When the agent's thread is started, the closure is applied to an input stream[1] and produces an output stream. The output stream is continually forced and the messages are then sent to the target agents. If the output stream becomes `[]` then the agent terminates and its thread dies.

An `OperatingSystem` agent implements system messages. It continually monitors its queue and dispatches on the name of the messages as they arrive. Asynchronous messages are handled by `async`. Synchronous messages are handled by `sync`. The class shows the implementation of messages `print` and `new`.

Figure 11 shows part of the data structures occurring in an EBG run-time system. The table `agents` associates agent identifiers with agents. The closure of a functional agent refers to the agent's identifier and message stream via its environment containing bindings for variables.

[1]The 6 arguments described in section 4 have been simplified here for the purposes of exposition.

6 Conclusion and Related Work

The long term goal of this work is to provide a programming environment that offers the advantages of both FP and Java. This paper has described the design and implementation of Agents in EBG that provide a programming interface between lazy higher-order functions and multi-processing. Agents have been implemented in EBG and current plans include using agents as part of a proposed EBG development environment written in EBG and to extend agents with facilities for networking.

The stream-based model of agents developed for EBG is based on existing work which aims to provide program state, multi-processing and interactive features in non-strict functional languages [Wad90] [Tho90] [Car98]. Agents offer *lightweight* processes and therefore the constructs in the agent calculus are limited (by the type system) as to where they occur and (by CPS) when they are executed. Other approaches to processes in non-strict FP, *e.g.* [Hal98], offer fine grain parallelism at all levels of a program using **par** and **seq** expressions. The design of Agents in EBG has been presented *computationally* using a term equivalence relation and a virtual machine. An alternative approach uses a process algebra as the semantics for EBG agents by translating an extended λ-calculus to the π-calculus [Mil93] [San99] [Cla99b].

There are a number of languages, currently in development, that aim to offer the advantages of both FP and Java. MLJ [Ben98] translates Standard ML to the Java Virtual Machine language and [Bot98a] [Bot98b] compiles Scheme to the Java VM. Both SML and Scheme are strict languages, but some of the issues in compilation are the same as EBG, for example the use of the abstract class `Closure`. Wakeling [Wak97] describes how Haskell can be compiled to the Java Virtual Machine running an implementation of the G-machine. EBG is a simpler language than Haskell and uses a single stack (the Java VM run-time stack) whereas the G-machine uses a pointer stack that is reportedly a problem when implemented in the Java VM as a large array [Wak98]. Pizza [Ode97] and GJ [Bra98] aim to provide the benefits of parametric types by extending Java although they do not address lazy evaluation and higher-order functions.

References

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

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

[Arn98] Arnold, K. & Gosling J. (1998): *The Java Programming Language.* Addison-Wesley.

[Ben98] Benton, N., Kennedy, A. & Russell, G. (1998): Compiling Standard ML to Java Bytecodes. In the proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming, Baltimore.

[Bot98a] Bothner P. (1998): Kawa - Compiling Dynamic Languages to the Java VM. Presented at the 1998 Usenix Conference in New Orleans.

[Bot98b] Bothner P. (1998): Kawa: Compiling Scheme to Java. Presented at the 1998 Lisp Users Conference in Berkeley, CA.

[Bra98] Brache G., Odersky M., Stoutamire D. & Wadler P. (1998): Making the future safe for the past: Adding Genericity to the Java Programming Language. In the proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, (OOPSLA 98).

[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.

[Cla99a] Clark, A. N. (1999): EBG: A Lazy Functional programming Language Implemented on the Java Virtual Machine. Technical Report submitted to the Computer Journal.

[Cla99b] Clark, A. N. (1999): Specification and Implementation of a Multi-Agent Calculus based on Higher-Order Functions. Technical Report.

[Fie89] Field, A. J. & Harrison, P. G. (1989): *Functional Programming.* Addison-Wesley Publishing Company.

[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.

[Hal98] Hall, J. G, Baker-Finch, C., Trinder, P. & King, D. J. (1998): Towards an Operational Semantics for a Parallel Non-strict Functional Language. In the proceedings of the International Workshop on the Implementation of Functional Languages, IFL 98.

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

[Lan64] Landin, P. J. (1964): The Mechanical Evaluation of Expressions. The Computer Journal, 6, pp. 308 – 320.

[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.

[Ode97] Odersky M. & Wadler P. (1997): Pizza into Java: Translating theory into practice. Symposium on Principles of Programming Languages, pp 146 – 159.

[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.

[Wak97] Wakeling, D. (1997): A Haskell to Java Virtual Machine Code Compiler. In the proceedings of the 9th International Workshop on the Implementation of Functional Languages, Springer Verlag, 1997, pp. 39 – 52, LNCS 1467.

[Wak98] Wakeling, D. (1998): Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine. In the proceedings of the 1998 Conference on Programming Languages, Implementations, Logics and Programs (PLILP 98). Springer Verlag 1998 pp. 335 – 352, LNCS 1490.

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