NeXeme: A Distributed Scheme Based on Nexus

Luc Moreau*, David De Roure*, and Ian Foster**

Abstract. The remote service request, a form of remote procedure call, and the global pointer, a global naming mechanism, are two features at the heart of Nexus, a library for building distributed systems. NeXeme is an extension of Scheme that fully integrates both concepts in a mostly-functional framework, hence providing an expressive language for distributed computing. This paper presents a semantics for this Scheme extension, and also describes a NeXeme implementation, including its distributed garbage collector.

1 Introduction

Scheme [20] is a mostly-functional language, i.e., it is a fully functional language that also supports imperative notions such as assignments and continuations, for efficiency and expressivity reasons. We believe that a distributed extension of such a language requires a mechanism to invoke functions remotely, so that distribution becomes part of the most fundamental operation of the language. Such an approach is also adopted by languages such as Obliq [2], Java + RMI [10, 22], and Compositional C++ [4], in which methods can be invoked remotely.

Nexus [7], a library for building distributed systems, has two salient features: a *remote service request* is a form of remote procedure call [1], and *global pointers* provide for global naming in a distributed environment. By offering a functionality close to remote function invocation, Nexus is a suitable building block for our distributed language. Furthermore, when designing a distributed version of Scheme, our concerns were portability and potential use of high-performance hardware or protocols (e.g. supercomputers, ATM, UDP). Nexus also addresses these concerns as it runs on a variety of platforms and protocols.

NeXeme integrates the Nexus approach, with its remote service requests and global pointers, into a mostly functional language. The result is a novel distributed programming language that offers expressivity, development ease, and automatic memory management (via a distributed garbage collector). NeXeme provides powerful abstractions for controlling distribution while remaining computationally efficient. We believe that NeXeme is an excellent medium for implementing other forms of parallelism such as communication channels [21, 9] and futures [12, 14]. It is also an ideal platform for developing distributed symbolic applications, based for example on distributed mobile agents.

In this paper, we formalise the concepts of remote service request and global pointers. To this end, in Section 3, we present a formal semantics for a simplified version of NeXeme, called *Idealised NeXeme*. The semantics is operational as it defines a mechanical way of evaluating NeXeme programs on an abstract machine. In Section 4, we describe the NeXeme implementation. We compare and discuss our approach with related work in Section 5. More information on NeXeme is available at the following URL [15].

2 The Nexus Architecture

Nexus [7] is structured in terms of five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. A computation executes on a set of *nodes*

* This research was supported in part by EPSRC grant GR/K30773. Authors' address: Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ. United Kingdom. E-mail: (L.Moreau,dder)@ecs.soton.ac.uk.

** Author's address: Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. E-mail: foster@mcs.anl.gov.

and consists of a set of *threads*, each executing in an address space called a *context*. (For the purposes of this article, it suffices to assume that a context is equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context.

The *global pointer* (GP) provides a global name space for objects, while the *remote service request* (RSR) is used to initiate communication and invoke remote computation. A GP represents a communication endpoint: that is, it specifies a destination to which a communication operation can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between nodes by including it in an RSR. A GP can be thought of as a capability granting rights to operate on the associated endpoint.

Practically, an RSR is specified by providing a global pointer, a handler identifier, and a data buffer, in which data are serialised. Issuing an RSR causes the data buffer to be transfered to the context designated by the global pointer, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and pointed specific data are available to the RSR handler.

The remote service request mechanism allows point-to-point communication, remote memory access, and streaming protocols to be supported within a single framework.

3 The Essence of Remote Service Requests

In this section, we present an operational semantics for remote service requests. For the sake of clarity and simplicity, we consider a subset of NeXeme, called idealised NeXeme. Table 1 summarises its different data structures, their representation, their constructors, and the operations permitted on them. (Programming examples may be found in [15].) The semantics was designed also to model non-functional languages like C; for example, full closures and first-class continuations are present in the semantics, but are not required in the actual implementation.

name	representation	constructors	operations
pair	$(\mathsf{cons}\ V_1\ V_2)$	cons	car, cdr
box	$\langle \mathsf{bx}\ \alpha, s\rangle$	makeref	deref, setref
lock	$\langle \mathsf{lk}\ \alpha,\ s\rangle$	make-lock	lock, unlock
global pointer	$\langle \mathsf{gp}\ \alpha,\ s\rangle$	make-gp,remote	remote service request
closure	$\langle \mathsf{cl}\ \lambda\vec{x}.M, \rho\rangle$	$\lambda\vec{x}.M$	apply
continuation	$\langle \mathsf{co}\ \kappa\rangle$	callcc	apply

Table 1. Data Structures

The state space of the semantics appears in Figure 1. Distributed computations proceed inside a *world* composed of several *sites*[3]. Each site is identified by a unique name s and is composed of a set T of *tasks* sharing access to a *store* θ. A store associates locations with values. Sites communicate by exchanging *requests*; each site contains both an output queue O of requests to be sent, and an input queue I of requests to be handled. When arriving at a site, requests are dealt with by a handler contained in H, a handler table associating names with closures.

A task is an abstraction of a Nexus thread composed of a *computational state* and a name. A computational state is a configuration of the CEK abstract machine [5] designed to evaluate the sequential subset of the language. Two states are permitted: $\mathsf{Ev}\langle M, \rho, \kappa\rangle$ represents the evaluation of a term M, in the environment ρ, with a continuation κ; $\mathsf{Ret}\langle V, \kappa\rangle$ models the return of a value V to a continuation κ. In terms of implementation, the term being evaluated is the program counter, the continuation is the control stack, and the environment is the sequence of frames containing bindings.

[3] In the semantics, a site corresponds to the notion of *context* in Nexus; we adopt a different name because contexts are usually given a different meaning in semantics [5].

$$
\begin{array}{lll}
\mathcal{W} & ::= \{m_1, \ldots, m_n\} & \text{(World)} \\
m \in \mathcal{M} & ::= \langle T, \theta, s, H, I, O \rangle & \text{(Site)} \\
t \in Task & ::= \langle C, \tau \rangle & \text{(Task)} \\
\theta \in Store & ::= \{(\alpha_1\ V_1) \ldots (\alpha_n\ V_n)\} & \text{(Store)} \\
s \in \mathcal{S} & = \{s_1, s_2, \ldots\} & \text{(Site Name)} \\
\tau \in \mathcal{T} & = \{\tau_1, \tau_2, \ldots\} & \text{(Task Name)} \\
 & & \\
C \in CoSt & ::= \mathsf{Ev}\langle M, \rho, \kappa \rangle & \text{(Computational State)} \\
 & \quad \mid \mathsf{Ret}\langle V, \kappa \rangle & \\
\rho \in Env & ::= \{(x_1\ V_1)\ \ldots (x_n V_n)\} & \text{(Environment)} \\
I & = \{R_1, \ldots, R_n\} & \text{(Input Queue)} \\
O & = \{R_1, \ldots, R_n\} & \text{(Output Queue)} \\
T & = \{t_1, \ldots, t_n\} & \text{(Tasks Set)} \\
H \in Hndl & ::= \{(str_1\ f_1) \ldots (str_n\ f_n)\} & \text{(Handlers)}
\end{array}
$$

Free Task Name:
$FTN(T) = \{\tau | \langle C, \tau \rangle \in T\}$
Free Site Name:
$FSN(\mathcal{W}) = \{s | \langle T, \theta, s, H, I, O \rangle \in \mathcal{W}\}$

$$
\begin{array}{r}
\delta(\mathsf{cons}, V_1, V_2) = (\mathsf{cons}\ V_1\ V_2) \\
\delta(\mathsf{car}, (\mathsf{cons}\ V_1\ V_2)) = V_1 \\
\delta(\mathsf{cdr}, (\mathsf{cons}\ V_1\ V_2)) = V_2 \\
\delta(+, \lceil n \rceil, \lceil m \rceil) = \lceil n + m \rceil
\end{array}
$$

$$
\begin{array}{lll}
M \in \Lambda & ::= V_s \mid (M\ M\ \ldots) \mid (\mathsf{if}\ M\ M\ M) & \text{(Term)} \\
V_s \in SValue & ::= x \mid (\lambda \vec{x}.M) \mid str \mid c & \text{(Syntactic Value)} \\
V \in RValue & ::= f \mid str \mid c \mid p \mid b \mid l \mid gp \mid k \mid \tau & \text{(Runtime Value)} \\
x \in Vars & = \{x, y, z \ldots\} & \text{(User Variable)} \\
\vec{x} \in VVec & ::= x \mid x\ \vec{x} \mid .x \mid .\mathbf{0} & \text{(Var Vector)} \\
f \in Clo & ::= \langle \mathsf{cl}\ \lambda \vec{x}.M, \rho \rangle & \text{(Closure)} \\
str \in String & & \text{(String)} \\
c \in Const & ::= c_b \mid c_f & \text{(Constant)} \\
c_b \in BConst & = \{\mathsf{true}, \mathsf{false}, \mathsf{nil}, 0, 1, \ldots, \mathsf{void}\} & \text{(Basic Constant)} \\
c_f \in FConst & = \{\mathsf{cons}, \mathsf{car}, \mathsf{cdr}, +, \mathsf{makeref}, \mathsf{deref}, & \text{(Functional Constant)} \\
 & \quad \mathsf{setref}, \mathsf{callcc}, \mathsf{make\text{-}gp}, \mathsf{remote}, \mathsf{rsr}, \mathsf{fork}, \mathsf{define\text{-}handler}, \mathsf{shutdown}\} & \\
p \in Pair & ::= (\mathsf{cons}\ V\ V) & \text{(Pair)} \\
b \in Box & ::= \langle \mathsf{bx}\ \alpha, s \rangle & \text{(Box)} \\
l \in Locks & ::= \langle \mathsf{lk}\ \alpha,\ s \rangle & \text{(Lock)} \\
gp \in GloP & ::= \langle \mathsf{gp}\ s,\ \alpha \rangle & \text{(Global Pointer)} \\
k \in Cont & ::= \langle \mathsf{co}\ \kappa \rangle & \text{(Continuation)} \\
\kappa \in CCode & ::= (\mathbf{init}) \mid (\kappa\ \mathbf{arg}\ \rho\ \langle V\ \ldots \bullet\ M\ \ldots \rangle) \mid (\kappa\ \mathbf{cond}(M, M, \rho)) & \text{(Cont. code)} \\
R \in Req & ::= \mathsf{Req}\langle s, sgp, so, \ldots \rangle & \text{(Request)} \\
so \in SO & ::= \langle serialised\ \ objects \rangle & \text{(Serialised Object)} \\
sgp \in SO & ::= \langle serialised\ \ global\ pointer \rangle & \text{(Serialised Object)}
\end{array}
$$

Environment Operations:

$$
\begin{array}{r}
\rho(x) = V \text{ if } (x\ V) \in \rho \\
\rho[x \leftarrow V] = (\rho \setminus \{(x\ V')\}) \cup \{(x\ V)\} \\
\text{if } (x\ V') \in \rho \\
\rho[x \leftarrow V] = \rho \cup \{(x\ V)\} \\
\text{if } x \notin DOM(\rho) \\
\rho[x\ \vec{x} \leftarrow V_1, V_2 \ldots] = \rho[x \leftarrow V_1][\vec{x} \leftarrow V_2 \ldots] \\
\rho[.\mathbf{0} \leftarrow] = \rho \\
\rho[.x \leftarrow] = \rho[x \leftarrow \mathsf{nil}] \\
\rho[.x \leftarrow V \ldots] = \rho[x \leftarrow listify(V \ldots)]
\end{array}
$$

Store Operations:

$$
\begin{array}{r}
\theta \uplus \{(\alpha\ V)\} = \theta \cup \{(\alpha\ V)\} \\
\text{with } \alpha \notin DOM(\theta) \\
\theta(\alpha) = V \text{ if } (\alpha\ V) \in \theta \\
\theta[\alpha := V] = (\theta \setminus \{(\alpha\ \theta(\alpha))\}) \cup \{(\alpha\ V)\}
\end{array}
$$

$$
\begin{array}{r}
listify(V_1) = (\mathsf{cons}\ V_1\ \mathsf{nil}) \\
listify(V_1, V_2 \ldots) = (\mathsf{cons}\ V_1\ listify(V_2 \ldots))
\end{array}
$$

Fig. 1. State Space of the CEKDS-machine

$$
\begin{array}{rll}
\mathsf{Ev}\langle(M\ M_1 ...), \rho, \kappa\rangle & \rightarrow_C \mathsf{Ev}\langle M, \rho, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \bullet, M_1, ...\rangle)\rangle & (\textit{operator}) \\
\mathsf{Ev}\langle \lambda\vec{x}.M, \rho, \kappa\rangle & \rightarrow_C \mathsf{Ret}\langle(\mathsf{cl}\ \lambda\vec{x}.M, \rho), \kappa\rangle & (\textit{lambda}) \\
\mathsf{Ev}\langle c, \rho, \kappa\rangle & \rightarrow_C \mathsf{Ret}\langle c, \kappa\rangle & (\textit{constant}) \\
\mathsf{Ev}\langle x, \rho, \kappa\rangle & \rightarrow_C \mathsf{Ret}\langle \rho(x), \kappa\rangle & (\textit{variable}) \\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle V_1, ..., \bullet, M, M_1, ...\rangle)\rangle & \rightarrow_C \mathsf{Ev}\langle M, \rho, (\kappa\ \mathbf{arg}\ \rho\ \ \langle V_1, ..., V, \bullet, M_1, ...\rangle)\rangle & \\
 & & (\textit{operand}) \\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle(\mathsf{cl}\ \lambda\vec{x}.M, \rho_1), V_1, ..., \bullet\rangle)\rangle & \rightarrow_C \mathsf{Ev}\langle M, \rho_1[\vec{x} \leftarrow V_1, ..., V], \kappa\rangle & (\textit{apply}) \\
\mathsf{Ret}\langle(\mathsf{cl}\ \lambda\vec{x}.M, \rho), (\kappa\ \mathbf{arg}\ \rho'\ \ \langle\bullet\rangle)\rangle & \rightarrow_C \mathsf{Ev}\langle M, \rho[\vec{x} \leftarrow], \kappa\rangle & (\textit{apply0}) \\
\mathsf{Ev}\langle(\mathsf{if}\ M\ M_1\ M_2), \rho, \kappa\rangle & \rightarrow_C \mathsf{Ev}\langle M, \rho, (\kappa\ \mathbf{cond}\ (M_1, M_2, \rho))\rangle & (\textit{predicate}) \\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{cond}\ (M, M_1, \rho))\rangle & \rightarrow_C \mathsf{Ev}\langle M_1, \rho, \kappa\rangle \quad \text{if } V = \mathsf{false} & (\textit{if else}) \\
 & \rightarrow_C \mathsf{Ev}\langle M, \rho, \kappa\rangle \quad \text{if } V \neq \mathsf{false} & (\textit{if then}) \\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle\mathsf{callcc}, \bullet\rangle)\rangle & \rightarrow_C \mathsf{Ret}\langle(\mathsf{co}\ \kappa), (\kappa\ \mathbf{arg}\ \rho\ \ \langle V, \bullet\rangle)\rangle & (\textit{capture}) \\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle(\mathsf{co}\ \kappa'), \bullet\rangle)\rangle & \rightarrow_C \mathsf{Ret}\langle V, \kappa'\rangle & (\textit{invoke}) \\
\mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle f, V_1, ..., \bullet\rangle)\rangle & \rightarrow_C \mathsf{Ret}\langle\delta(f, V_1, ..., V), \kappa\rangle & (\delta)
\end{array}
$$

Fig. 2. CEK Machine

The transitions between computational states in Figure 2 deal with the sequential subset of the language that does not involve the store. The transitions extend the CEK-machine transitions [5] by accepting n-ary applications ($M_1\ M_2\ \ldots$) and abstractions ($\lambda\vec{x}.M$) with variable number of arguments. Consequently the continuation code (κ **arg** $\rho\ \langle V_1, \ldots, V_n, \bullet, M_{n+2}, \ldots\rangle$) conveys the following meaning: the first n components of the application have already been evaluated and their values are $V_1, \ldots, V_n$; the next component, i.e., the $n+1$th, is being evaluated; the components $M_{n+2}, \ldots$ remain to be evaluated in the environment ρ.

Figures 3 to 6 deal with site transitions. According to rule (*sequential*), a site can perform a transition if it contains a task that can perform a transition of Figure 2. In rule (*fork*), the primitive fork creates a new task applying the closure received as argument and returns the new task name.

Figure 4 deals with operations related to boxes. The primitive makeref allocates a new location α, stores the value V in that location, and returns a new box object $\langle\mathsf{bx}\ \alpha, s\rangle$, pointing at the location α in the current site s. The primitive deref returns the content of the location associated with the box and the primitive setref changes the content of its associated location.

Rule (*make gp*) creates global pointers in the same way as (*makeref*) creates boxes. A remote service request has the form $(\mathsf{rsr}\ str\ gp\ M_1\ \ldots M_n)$, where *str* must evaluate to a string naming a handler to be called on the site that *gp* is pointing at, with the arguments obtained as values of $M_1, \ldots, M_n$. The value returned by a remote service request is the distinguished void value, but its effect is to add a *request* in the output queue of the current site. A request is not a first-class object, i.e., it is not part of the set of RValues, but it contains the destination site designated by *gp*, the serialised string, and the serialised values of $M_1, \ldots, M_n$.

Let us observe that the semantics does not specify the behaviour of the $\mathcal{S}erial$ and $\mathcal{D}eserial$ functions. Their purpose is to convert to and from a suitable format for transportation. The composition of these functions returns a result that is an isomorphic copy of its argument. (The specification is available from [15].)

If there is a request in the input queue of a site, rule (*handle*) deserialises its content, and applies the handler associated with *str* to the value indicated by the global pointer and the deserialised values. It is also worth noticing that the incoming request is handled

$$\langle \{\ \langle C, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle C_1, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle \quad \text{if } C \to_C^* C_1 \qquad (\textit{sequential})$$
$$\langle \{\ \langle \mathsf{Ret}\langle(\mathsf{cl}\ \lambda\vec{x}.M, \rho), (\kappa\ \mathbf{arg}\ \rho_1\ \ \langle \mathsf{fork}, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \tau_1, \kappa\rangle, \tau\rangle\ \} \cup \{\ \langle \mathsf{Ev}\langle M, \rho[\vec{x} \leftarrow], (\mathbf{init})\rangle, \tau_1\rangle\ \} \cup T, \theta, s, H, I, O\rangle \qquad (\textit{fork})$$
$$\quad \text{with } \tau_1 \notin FTN(T) \cup \{\tau\}$$

Fig. 3. Sequential and Parallel Evaluation

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{makeref}, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle(\mathsf{bx}\ \alpha, s\rangle, \kappa\rangle, \tau\rangle\ \} \cup T, \theta \uplus \{(\alpha\ V)\}, s, H, I, O\rangle \qquad (\textit{makeref})$$
$$\langle \{\ \langle \mathsf{Ret}\langle(\mathsf{bx}\ \alpha, s\rangle, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{deref}, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \theta(\alpha), \kappa\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle \qquad (\textit{deref})$$
$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{setref}, (\mathsf{bx}\ \alpha, s\rangle, \bullet)\rangle), \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, \tau\rangle\ \} \cup T, \theta[\alpha := V], s, H, I, O\rangle \qquad (\textit{setref})$$

Fig. 4. Boxes

$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{make\text{-}gp}, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle(\mathsf{gp}\ \alpha,\ s\rangle, \kappa\rangle, \tau\rangle\ \} \cup T, \theta \uplus \{(\alpha\ V)\}, s, H, I, O\rangle \qquad (\textit{make gp})$$
$$\langle \{\ \langle \mathsf{Ret}\langle V_n, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{rsr}, str, \langle \mathsf{gp}\ \alpha,\ s\rangle, V_1, ..., \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s_1, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, \tau\rangle\ \} \cup T, \theta, s_1, H, I, \{\ \mathsf{Req}\langle s, sgp, sstr, so_1, ..., so_n\rangle\ \} \cup O\rangle \qquad (\textit{rsr})$$
$$\quad \text{if } \mathit{Serial}(s_1, \theta, \langle(\mathsf{gp}\ \alpha,\ s\rangle, str, V_1, ..., V_n\rangle) = \langle sgp, sstr, so_1, ..., so_n\rangle$$
$$\langle T, \theta, s, H, \{\ \mathsf{Req}\langle s, sgp, str, so_1, ..., so_n\rangle\ \} \cup I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ev}\langle M, \rho[\vec{x} \leftarrow \theta(\alpha), V_1, ..., V_n], (\mathbf{init})\rangle, \tau\rangle\ \} \cup T, \theta_1, s, H, I, O\rangle \qquad (\textit{handle})$$
$$\quad \text{if } H(str) = \langle \mathsf{cl}\ \lambda\vec{x}.M, \rho\rangle,$$
$$\quad\ \mathit{Deserial}(s, \theta, \langle sgp, so_1, ..., so_n\rangle) = \langle\langle(\mathsf{gp}\ \alpha,\ s\rangle, V_1, ..., V_n\rangle, \theta_1\rangle$$
$$\quad \text{with } \tau \notin FTN(T)$$
$$\langle \{\ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{define\text{-}handler}, str, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, \tau\rangle\ \} \cup T, \theta, s, H \uplus \{(str\ V)\}, I, O\rangle \qquad (\textit{define handler})$$

Fig. 5. Global Pointers and Remote Service Requests

$$\langle \{\ \langle \mathsf{Ret}\langle \mathsf{make\text{-}lock}, (\kappa\ \mathbf{arg}\ \rho\ \ \langle\bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle(\mathsf{lk}\ \alpha,\ s\rangle, \kappa\rangle, \tau\rangle\ \} \cup T, \theta \uplus \{(\alpha\ \mathsf{true})\}, s, H, I, O\rangle \qquad (\textit{make-lock})$$
$$\langle \{\ \langle \mathsf{Ret}\langle(\mathsf{lk}\ \alpha,\ s\rangle, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{lock}, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, \tau\rangle\ \} \cup T, \theta[\alpha := \mathsf{false}], s, H, I, O\rangle \quad \text{if } \theta(\alpha) = \mathsf{true} \qquad (\textit{acquire})$$
$$\langle \{\ \langle \mathsf{Ret}\langle(\mathsf{lk}\ \alpha,\ s\rangle, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{unlock}, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle$$
$$\quad \to_m \langle \{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, \tau\rangle\ \} \cup T, \theta[\alpha := \mathsf{true}], s, H, I, O\rangle \qquad (\textit{release})$$

Fig. 6. Mutual Exclusion

$$\{m_1\} \cup \mathcal{W}$$
$$\rightarrow_{\mathcal{W}} \{m_2\} \cup \mathcal{W} \quad \text{if } m_1 \rightarrow_m m_2 \qquad (\textit{site})$$
$$\{\langle T, \theta, s, H, I, \{\ \mathsf{Req}\langle s, req\rangle\ \} \cup O\rangle \cup \mathcal{W}$$
$$\rightarrow_{\mathcal{W}} \{\langle T, \theta, s, H, \{\ \mathsf{Req}\langle s, req\rangle\ \} \cup I, O\rangle \cup \mathcal{W} \qquad (\textit{local request})$$
$$\{\langle T_0, \theta, s_0, H_0, I_0, \{\ \mathsf{Req}\langle s_1, req\rangle\ \} \cup O_0\rangle, \langle T_1, \theta_1, s_1, H_1, I_1, O_1\rangle\} \cup \mathcal{W}$$
$$\rightarrow_{\mathcal{W}} \{\langle T_0, \theta, s_0, H_0, I_0, O_0\rangle, \langle T_1, \theta_1, s_1, H_1, \{\ \mathsf{Req}\langle s_1, req\rangle\ \} \cup I_1, O_1\rangle\} \cup \mathcal{W}$$
$$(\textit{migrate request})$$
$$\{\langle\{\ \langle \mathsf{Ret}\langle(\mathsf{gp}\ \alpha,\ s\rangle, (\kappa\ \mathbf{arg}\ \rho\ \ \langle \mathsf{remote}, \text{“host”}, \langle \mathsf{cl}\ \lambda\vec{x}.M, \emptyset\rangle, \bullet\rangle)\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle\} \cup \mathcal{W}$$
$$\rightarrow_{\mathcal{W}} \{\langle\{\ \langle \mathsf{Ret}\langle(\mathsf{gp}\ \alpha_1,\ s_1\rangle, \kappa\rangle, \tau\rangle\ \} \cup T, \theta, s, H, I, O\rangle, m\} \cup \mathcal{W} \qquad (\textit{remote})$$
$$\text{with } m = \langle\{\ \langle \mathsf{Ev}\langle M, \emptyset[\vec{x} \leftarrow \langle \mathsf{gp}\ \alpha,\ s\rangle], (\mathbf{init})\rangle, \tau\rangle\ \}, \theta_1, s_1, \emptyset, \emptyset, \emptyset\rangle$$
$$\text{with } \theta_1 = \{(\alpha_1\ \mathsf{nil})\},\ s_1 \notin FSN(\mathcal{W}) \cup \{s\}$$

Fig. 7. Other Distribution Aspects

asynchronously and that the handler is executed in a new thread. Optimistic active-messages [25] show that the thread could be created lazily [13] and still produce good performance. According to rule (*define handler*), the primitive define-handler allows a program to add new handlers in a given site.

Idealised NeXeme offers some primitives to implement critical sections. A *lock* is a datastructure that can be acquired by at most one task using the primitive lock, and that can be released by the primitive unlock. In Figure 6, a busy-wait implementation of locks relies on the fact that transition rules are atomic.

Figure 7 displays transitions between worlds. Rule (*site*) shows that a world can perform a transition, if there is a site which can perform a transition of Figures 3 to 6. In addition, rules (*local request*) and (*migrate request*) deal with request transfers, and rule (*remote*) is used to create a new site in a world.

Creation of a remote site is always a delicate problem in a distributed system. Frequently, libraries for distribution assume an SPMD approach and use operating-system primitives to startup remote processes. We decided not to model NeXeme very closely, as the semantics would have been unsatisfactorily complicated. Instead, Idealised NeXeme provides a primitive to create a new remote site and to initiate a computation on that site. This initial computation is essential, because it allows the user to install initial handlers for remote service requests. In addition, it makes it possible to synchronise both sites so that the initial site is kept informed when the new site is initialised and is ready to perform some processing.

A remote site is created by the primitive remote which takes the following arguments: the name of a host, a closure representing the initial computation to be done, and a global pointer that can be used by the remote site to notify the end of initialisation by a remote service request. The value returned by the primitive remote is a global pointer pointing at the new remote site. In order to be implementable in languages without closures, like C, rule (*remote*) requires the closure to have an empty environment. As a result, the closure can be encoded as a pointer to its entry point. Let us observe that the remote site should be executed on “host”.

Having defined a transition relation between worlds, we can now define an evaluation relation associating programs with values. Detecting termination in a distributed system is a difficult matter in the general case. In Idealised NeXeme, we assume there exists a primitive shutdown that the program has to call on the origin site; it terminates the computation and returns its argument.

Definition 1 (Eval) Let M be a closed term. An *initial task* is of the form $t_i =$

$\langle \mathsf{Ev}(M, \emptyset, ((\mathbf{init}))), \tau \rangle$, while a *final task* is of the form $t_f = \langle \mathsf{Ret}(V, (\kappa\ \mathbf{arg}\ \rho\ \langle \mathsf{shutdown}, \bullet \rangle)), \tau' \rangle$. We have that:

$$\mathrm{eval}(M) = V \text{ if } \langle \{\!\{\ t_i\ \}\!\}, \emptyset, s_0, \emptyset, \emptyset, \emptyset \rangle \rightarrow_{\mathcal{W}} \{\langle \{\!\{\ t_f\ \}\!\} \cup T, \theta, s_0, H, I, O \rangle\} \cup \mathcal{W}.$$

□

4 The Implementation

4.1 General Description

Figure 8 displays the organisation of the NeXeme implementation. As Nexus is multi-threaded, we had to adopt a thread-safe garbage collector. The public domain Boehm-Weiser's [11] garbage collector supports various OS-level threads and is also part of the PPCR portable common runtime system [26]. On each platform, Nexus is recompiled against the garbage collector (abbreviated gc) and the suitable thread package. The executable is generated by linking the Scheme system with the resulting libraries Nexus, gc, and threads.

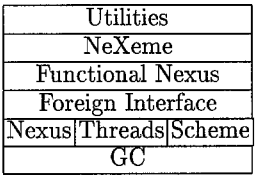

Fig. 8. Organisation of NeXeme

Primitives of the Nexus library are made accessible through a foreign interface definition. Let us note that some Nexus primitives require procedures as argument (for instance, thread creation or callbacks for handlers). In order to integrate properly Nexus with Scheme, we modified Nexus to support callbacks to Scheme functions. While the foreign interface defines a similar API as the Nexus library, the "functional Nexus" layer provides a more functional interface to Nexus; for instance, results are returned by functions, errors are signalled by exceptions, and memory is managed automatically. The NeXeme layer offers a functional version of remote service requests as described in Section 3. Finally, a library defining a set of utilities provides other paradigms for distribution like futures, communication channels, or farm processing.

4.2 Distributed Garbage Collection

NeXeme has a distributed garbage collector that takes care of memory management automatically. Our working hypotheses, provided by Nexus, are a reliable message-passing and a FIFO ordering[4] of messages between two sites.

Each site relies on a thread safe, conservative, mark and sweep garbage collector [11]; conservativeness is required as Scheme data are passed to Nexus, written in C, and are pointed by Nexus data structures. In addition, NeXeme maintains two tables for each site. The *exit table*[5] associates each global pointer with the number of distinct remote copies of this pointer originating from the site. The *entry table* contains all global pointers received by a site, except those that point at itself. The exit table, but not the entry table, is a root of the local garbage collector. The role of the distributed collector it to update counters in a safe and consistent way. To this end, it relies on two types of *control* messages, called "*decrement*(gp)" and "*increment-decrement*(gp, s)", as described below.

[4] Nexus, as opposed to Idealised NeXeme, allows requests to be handled in a non-threaded manner; in such a case, Nexus guarantees the FIFO ordering of requests between two sites.

[5] Different names are found in the literature, such as entry/exit items, scions/stubs [17].

Reference counters are updated according to the *diffusion tree* [16] of global pointers. The first time a global pointer *gp* is serialised, an entry is added into the exit table of the current site with a counter set to 1; afterwards, for every serialisation, this counter is incremented. Symmetrically, the first time a global pointer *gp* is deserialised, it is added to the entry table (if it points at a remote host); if it is already present, NeXeme sends a decrement message for this global pointer, "*decrement*(*gp*)" to the site that sent the remote service request. When a site receives a message "*decrement*(*gp*)", the counter of *gp* in the exit table is decremented. Once the counter reaches zero, the global pointer may be removed from the table; only then, the pointer itself and the local data may be reclaimed, if no longer accessible.

In Nexus, a global pointer contains the site it is pointing at, and not the site it is arriving from. Therefore, once a global pointer *gp* pointing at a site s_1 becomes inaccessible on a site s_2, s_2 sends a message "*decrement*(*gp*)" to s_1. This naive implementation of the reference counter technique is sound if causality is preserved in the system [16]. This is ensured by a reorganisation of the diffusion tree, as explained in the following scenario illustrated in Figure 9.

Let us consider that *gp*, a global pointer pointing to address α in s_1, is migrated to s_2, and then migrated to s_3. Figure 9 part 1 shows that the reference counters in s_1 and s_2 are equal to 1, meaning that there are $1 + 1 = 2$ active remote references of *gp*. Once s_3 deserialises *gp*, the global pointer received from s_2, a reorganisation can be initiated, as illustrated in Figure 9 part 2. *(i)* Site s_3 sends an "*increment-decrement*(*gp*, s_2)" message to s_1; *(ii)* when the message is received by s_1, the counter for *gp* is incremented on s_1; *(iii)* afterwards, a "*decrement*(*gp*)" message is sent to s_2; *(iv)* when the message is received, the counter for *gp* is decremented on s_2.

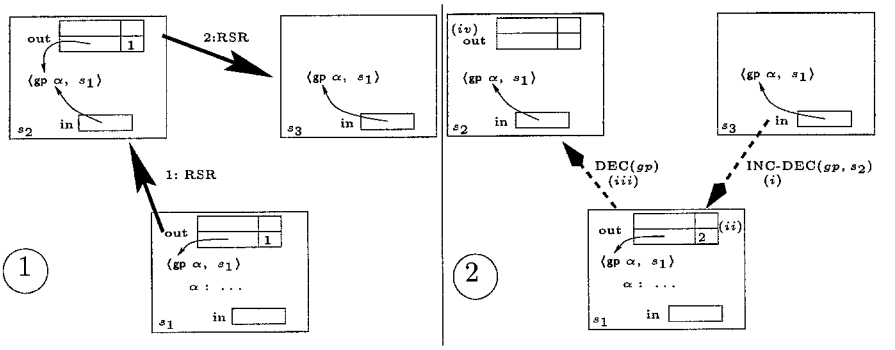

Fig. 9. Distributed Reference Counters (1) after Remote Service Requests (2) after Control Messages

Race conditions are avoided between s_1 and s_3 by giving priority to "increment-decrement" over "decrement" messages. As a result, we have the following invariant. For every global pointer *gp* pointing to site *s*:

$$\sum_{site} counter(gp) = \sum_{site \neq s} in(gp) + \sum Value(message\ for\ gp)$$

with $Value(decrement(gp)) = 1$ and $Value(increment\text{-}decrement(gp, s_i)) = 0$.

The simplicity and portability of the solution is unfortunately counter-balanced by its inability to collect distributed cycles. Our approach differs from "Indirect Reference Counting" [16] because it can reclaim "*zombie*" pointers, i.e., *gp* can be freed on s_2 even though it remains active on s_3; other techniques to reduce the length of indirection chains

are described in [17]. Several optimisations are possible. First, control messages of the same type may be grouped in a single message. Second, an "*increment-decrement*(gp, s_2)" to be sent to s_1, followed by a "*decrement*(gp)" to the same destination, may be replaced by a "*decrement*(gp)" to s_2.

4.3 Utilities

The utilities layer offers a number of services that we briefly describe in this section. *Remote invocation of a procedure* sends a locally-available closure to a remote site where it is invoked on some arguments; as in a remote service request, no result is returned by this operation. Not only does *remote invocation of a function* transfer and call a function on some arguments, but also it returns the result to the site that initiated the operation. Communication channels [21, 9], and futures [12] are also provided.

"Farm-processing" versions of the various remote call utilities are provided that can be used when a list of global pointers is passed as an argument; these versions map the corresponding operation onto each element of the list. NeXeme is also able to detect when a site becomes idle so that lazy task creation [13] and task stealing can be implemented easily.

5 Discussion and Related Work

The Nexus [7] philosophy is derived from *active messages* [24], where each message contains at its head the address of a user-level handler executed on message arrival. Nexus has also been used to build several other distributed languages. In Fortran M [6], program modules, called processes, communicate via explicitly-declared channels. The language nPerl [8] is a distributed extension of Perl which provides a remote procedure call; unlike NeXeme, higher-order functions and continuations are not available. Compositional C++ [4] supports global pointers that can refer to objects on remote machines, and defines usual operations on them; also, CC+ offers single assignment objects. Currently, NeXeme uses Nexus 3.0; Nexus 4.0 provides EZ Nexus a more abstract interface to remote service requests, which is similar to the one described here.

Boxes allow the programmer to define mutable data structures, for example cyclic data structures. If a cyclic data structure is passed as an argument to a remote service request, an isomorphic copy is generated on the remote site. NeXeme does not prevent the user from mutating each copy separately, as they are now independent entities. However, using global pointers, any form of consistency can be programmed on top of NeXeme, such as simulated shared memory [14] or causally-coherent memory [18].

There exist other distributed implementations of functional languages with *explicit* notations for parallelism or distribution. DMeroon [18] offers a distributed memory model which enforces coherence based on causality, while ICSLAS [19] adds to DMeroon transparent remote execution. Kali Scheme [3] also offers a form of active messages, on top of which higher-level primitives are defined; the semantics of Kali Scheme could easily be defined in terms of Idealised NeXeme. Distributed versions of SML-style languages are usually based on communications channels [21, 9].

6 Conclusion

This paper presents NeXeme, a distributed dialect of Scheme, based on remote service requests and global pointers provided by the library for distribution Nexus. This paper describes the semantics but also the implementation of this language. The functional interface to remote service requests of NeXeme is a perfect abstraction to build a Scheme with futures [14]; in addition, NeXeme is used for programming multimedia distributed applications.

7 Acknowledgement

Hans Boehm, Carl Hauser, Carl Kesselman, Andreas Kind, Christian Queinnec, Manuel Serrano, and Steve Tuecke gave useful comments and advice for the implementation of NeXeme.

Bibliography

[1] A. D. Birell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[2] L. Cardelli. A Language with Distributed Scope. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 286–297, San-Francisco, California, January 1995.

[3] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.

[4] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*, pages 281–313. MIT Press, 1993.

[5] M. Felleisen and D. P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).

[6] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. *J. of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. of Parallel and Distributed Computing*, 37:70–82, 1996.

[8] I. Foster and R. Olson. A Guide to Parallel and Distributed Programming in nPerl. Mathematics and Computer Science Division, October 1995.

[9] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *Int. J. of Parallel Programming*, 18(2):121–160, 1989.

[10] J. Gosling, G. Steele, and B. Joy. *The Java Language Specification*. Addison-Wesley, 1996.

[11] H.-J.Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[12] R. Halstead, Jr. Parallel Symbolic Computing. *IEEE Computer*, pages 35–43, Aug. 1986.

[13] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy Task Creation : a Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[14] L. Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, LNCS 1123, pages 615–624, 1996.

[15] NeXeme Home Page. `http://www.ecs.soton.ac.uk/~lavm/NeXeme`.

[16] J. M. Piquer. Indirect Distributed Garbage Collection: Handling Object Migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–647, 1996.

[17] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Int. Workshop on Memory Management (IWMM95)*, LNCS 986, pages 211–249, 1995.

[18] C. Queinnec. DMEROON: a Distributed Class-based Causally-coherent Data Model: Preliminary Report. In *Parallel Symbolic Languages and Systems.*, LNCS 1068, 1995.

[19] C. Queinnec and D. De Roure. Design of a Concurrent and Distributed Language. In *Parallel Symbolic Computing: Languages, Systems and Applications*, LNCS 748, pages 234–259, Boston, Massachussetts, October 1992.

[20] J. Rees and W. Clinger, editors. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[21] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, June 1992.

[22] Sun MicroSystems. Java Remote Method Invocation Specification, November 1996.

[23] P. van Roy, S. Haridi, P. Brand, and G. Smolka et al. Mobile Objects in Distributed Oz. Technical report, Swedish Institute of Computer Science, December 1996.

[24] T. von Eicken, D. E. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication & Computation. In *Proceedings of the 19th symposium on Computer Architecture*, pages 256–266, 1992.

[25] D. A. Wallach, W. C. Hsieh, K. Johnson, M. F. Kaashoek, *et al.* Optimistic active messages: A mechanism for scheduling communication with computation. In *5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '95)*, 1995.

[26] M. Weiser, A. Demers, and C. Hauser. The Portable Common Runtime Approach to Interoperability. In *ACM Symp. on Operating System Principles*, pages 114–122, Dec. 1989.