Behavioural types for non-uniform memory accesses

Juliana Franco and Sophia Drossopoulou

Imperial College London, United Kingdom

Abstract

Concurrent programs executing on NUMA architectures consist of concurrent entities (e.g. threads, actors) and data placed on different nodes. Execution of these concurrent entities often reads or updates states from remote nodes. The performance of such systems depends on the extent to which the concurrent entities can be executing in parallel, and on the amount of the remote reads and writes.

We consider an actor-based object oriented language, and propose a type system which expresses the topology of the program (the placement of the actors and data on the nodes), and an effect system which characterises remote reads and writes (in terms of which node reads/writes from which other nodes). We use a variant of ownership types for the topology, and a combination of behavioural and ownership types for the effect system.

1 Introduction

A prevalent paradigm in high performance machines is NUMA (non uniform memory access) systems, e.g., the AMD Bulldozer server[1]. NUMA systems have many *nodes* which contain processors and memory; Figure 1 shows the common NUMA structure. The nodes are connected with the other nodes through a system bus that allows processes running on a specific node to access the memory of the other nodes.

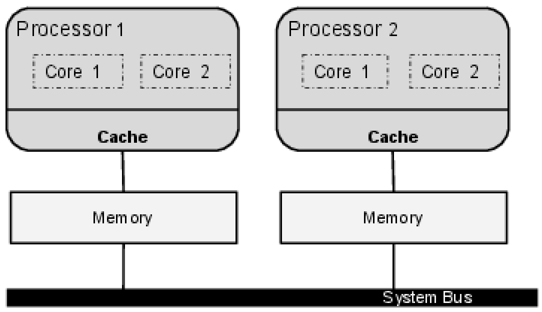

Figure 1: NUMA system [13].

Memory access is either local, i.e. accessing memory in the local node, or remote, i.e. accessing memory of remote nodes. Remote accesses require requests to the system bus, and are thus more expensive than local accesses. Moreover, different remote accesses do not necessarily have the same cost (the time to obtain/write data in memory). Therefore, to characterize the communication (read/write) costs of a concurrent program, we need to know its topology (the placement of the actors and data on the nodes), and a characterisation of the reads and writes across nodes.

In this work we consider a concurrent language based on actors (or active objects) and objects [5], which we call $\mathcal{L}_{numa}$, a language where, for the sake of simplicity, mutually recursive (synchronous and asynchronous) method invocations with communication are assumed to be not allowed and all the active objects must be created in the main class.

We develop a variant of ownership types [6] to express the location of actors and of data. In particular, we propose two levels of abstraction: classes have ownership (location) parameters, the main program defines the abstract locations and creates objects in these abstract locations; and at runtime the abstract locations are mapped to nodes (cf. Appendix C). We also propose a combination of behavioural and ownership types to characterise the interactions (reads, writes and messages sent) among objects located in different nodes.

Ownership types [4, 6, 12] were first introduced to statically describe the heap topology. Here we introduce ownership-like annotations to describe the system topology, that is, its nodes and where threads are running and data is allocated. Behavioural types [2, 8, 9, 14, 18] are usually used to describe and statically, or dynamically, verify patterns of interaction between processes/threads/participants of concurrent and parallel computations. Here we present a type system that allows the programmer to specify the interactions among objects located in different nodes, and therefore we abstract the communication made through the system bus.

Outline. This paper is organised as follows: Section 2 introduces the syntax of $\mathcal{L}_{numa}$, Section 3 gives the operational semantics, Section 4 presents the typing rules, and Section 4 shows properties of $\mathcal{L}_{numa}$, and finally Section 6 concludes. Several definitions are given in the appendix.

2 Syntax

Figure 2 presents the syntax of $\mathcal{L}_{numa}$. A program consists of a set of class declarations representing actors and passive objects. The use of the keyword active in a class declaration indicates that the class represents actors. Passive objects are similar to ordinary Java objects while actors have all the properties of passive objects, but in addition also have their own execution thread and may send messages to other actors. As in actor-based languages, messages are stored in private queues. A more detailed definition can be found in [5].

$$
\begin{aligned}
P \in \mathit{Program} &::= \mathit{Cd}^* \\
\mathit{Cd} \in \mathit{ClassDecl} &::= [\mathsf{active}]\ \mathsf{class}\ C\langle \overline{p}^+ \rangle\ \overline{\mathit{Fd}}\ \overline{\mathit{Md}} \\
\mathit{Fd} \in \mathit{FieldDecl} &::= f : T \\
\mathit{Md} \in \mathit{MethodDecl} &::= \mathsf{def}\ m(x : T) :\ T\ \mathsf{as}\ b\ \{e\} \\
ot \in \mathit{OwnershipType} &::= C\langle \overline{l}^+ \rangle \\
T \in \mathit{Type} &::= \mathsf{bool} \mid \mathsf{nil} \mid \boxed{\mathsf{int}} \mid ot \\
l \in \mathit{Location} &::= p \mid L \\
val \in \mathit{Value} &::= \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \\
var \in \mathit{Variable} &::= x \mid \mathsf{this}
\end{aligned}
$$

$$
\begin{aligned}
e \in \mathit{Expr} &::= var \mid val \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
&\mid e.m(e) \mid e!m(e) \mid e.f \\
&\mid e.f = e \mid \mathsf{new}\ ot \\
&\mid \mathsf{for}\ i\ \mathsf{in}\ n_1..n_2\ \mathsf{do}\ e \\
&\mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \boxed{\mathsf{return}\ e} \\
\pi \in \mathit{RemAccess} &::= \mathsf{rd}(l, l) \mid \mathsf{wrt}(l, l) \mid \mathsf{msg}(l, l, m) \\
bop &::= \pi \mid \{b\ \mathsf{or}\ b\} \mid \mathsf{Loop}(n\colon b) \\
b \in \mathit{Behaviour} &::= \varepsilon \mid bop.b \mid [b, b]
\end{aligned}
$$

Figure 2: Syntax of classes and (behavioural) types. The boxed constructs are not user syntax. In the class declaration Cd the use of [ ] means that the keyword active is optional.

Each class, active or passive, is annotated with a set of location parameters $p_1, \ldots, p_n$ where p_1 represents the place where the instance of the class is allocated and $p_2, \ldots, p_n$ locations that can be used in the types of the rest of the class. The location parameters of the main class, $L_1, ..., L_n$, are abstractions of the concrete nodes, and at runtime will be mapped to concrete node identifiers.

A class declaration might have field and method declarations. A field declaration consists of a field identifier and its type; a method declaration consists of a method identifier, one parameter (variable and type), return type, behavioural type and an expression (method body). $\mathcal{L}_{numa}$ has the types bool, nil, and an ownership type $C\langle l_1, ..., l_n \rangle$ which represents objects located in l_1 that may contain references to objects in locations $l_2, ..., l_n$. The syntax of expressions is similar to other OO programming languages; note only the asynchronous method call (message sending), $e!m(e)$.

The most interesting part of the syntax is our treatment of behavioural types. We have basic operations, π, which are reading from a remote node ($\mathsf{rd}(l, l)$), writing to a remote node ($\mathsf{wrt}(l, l)$), and message sending ($\mathsf{msg}(l, l, m)$)—this has to be reflected in the behaviour, as it adds messages to queues in remote memory. For all of them the first location is where the expression is running and the second is the location where a read/write is made or a message sent. We also have types to describe conditional expressions, $\{b \ \mathsf{or}\ b\}$, (the two branches in the expression imply two branches in the type), and for-loops, $\mathsf{Loop}(n\colon b)$. A behavioural type, b, may be empty, ε, meaning that there is no "communication" across different nodes, the sequence of operations, $bop.b$, and two types in parallel, $[b, b]$, introduced by message sendings.

3 Semantics

We now describe the dynamic semantics of $\mathcal{L}_{numa}$. Nodes, $\mathcal{N}$, defined in Figure 3, aim to reflect NUMA nodes. Namely, a node in our formalism has an identifier, a heap with all the data allocated in it, and several execution threads *Ethread*. An execution thread belongs to an actor, and has a stack and an expression being executed. A heap is a mapping from addresses

$$
\begin{array}{rl}
\mathcal{N} \in \mathit{Node} = & \mathit{NodeId} \times \mathit{Heap} \times \overline{\mathit{EThread}} \\
\mathcal{T} \in \mathit{EThread} = & \mathit{Stack} \times \mathit{Expr} \\
h \in \mathit{Heap} = & \mathit{Addr} \rightarrow \mathit{Object} \\
\sigma \in \mathit{Stack} = & \mathit{Addr} \times \overline{\mathit{Frame}} \\
\varphi \in \mathit{Frame} = & \mathit{var} \rightarrow \mathit{value} \\
Q \in \mathit{Queue} ::= & \bullet \mid \emptyset \mid m(v) :: Q \\
\mathcal{L} \in \mathit{LocsMap} = & \mathit{LocId} \rightarrow \mathit{NodeId} \\
\kappa \in \mathit{NodeId} = & \mathbb{N}
\end{array}
\qquad
\begin{array}{rl}
o \in \mathit{Object} = & \mathit{ClassId} \times \overline{\mathit{NodeId}} \times \\
& (\mathit{FieldId} \rightarrow \mathit{value}) \times \mathit{Queue} \\
\alpha \in \mathit{Addr} = & \mathit{NodeId} \times \mathbb{N} \\
v \in \mathit{value} = & \mathit{val} \mid \mathit{Addr} \mid \mathsf{skip} \mid \mathsf{NPE} \\
E[] ::= & [\cdot] \mid [\cdot].m(e) \mid \alpha.m([\cdot]) \mid [\cdot]!m(e) \mid \alpha!m([\cdot]) \\
& \mid [\cdot].f \mid [\cdot].f = e \mid \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ e \mid \alpha.f = [\cdot] \\
& \mid \mathsf{if}\ [\cdot]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ e \mid \mathsf{return}\ [\cdot] \\
l \in \mathit{Location} ::= & \text{as before} \mid \kappa
\end{array}
$$

Figure 3: Dynamic Entities. We assume the existence of a map $\mathcal{L}$ that maps abstract locations (declared by the programmer in the main class) to NUMA node identifiers.

to (passive and active) objects. An object consists of a class identifier C, a sequence of node identifiers representing the actual location parameters, a mapping from field identifiers to their values, and a message queue, where the queue of a passive object is $\bullet$. An address, α, consists of a node identifier, $\kappa \in \mathit{NodeId}$, and an offset, $n \in \mathbb{N}$.

In our system, a configuration $\overline{\mathcal{N}}$ can be reduced to another configuration $\overline{\mathcal{N}}'$ either without any communication or implying a remote access from one of the nodes to another node. In the first case, the rule **[GsExec1]** should be applied, where only one node is involved in the reduction. In the second case the rule **[GsExec2]** should be used, where two nodes are involved in the reduction, as shown in Figure 4.

In the same way, expression reduction may result in accessing remote memory or not; therefore we divide the operational semantics rules as follows:

1. Expressions that do not access memory or send messages. These are defined in Figure 5.
2. Expressions that result in accesses to memory. These are defined in Figure 6 and are further divided in:
   (a) The access happens locally—only one node required.

$$\textbf{[GsExec1]}\quad \frac{\kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e'}{\overline{\mathcal{N}}, (\kappa, h, \overline{\mathcal{T}}, \langle \sigma, e\rangle) \xrightarrow{\pi} \overline{\mathcal{N}}, (\kappa, h', \overline{\mathcal{T}}, \langle \sigma', e'\rangle)}$$

$$\textbf{[GsExec2]}\quad \frac{\kappa_1, h_1, \sigma_1, e_1 \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1', \sigma_1', e_1' \parallel h_2'}{\overline{\mathcal{N}}, (\kappa_1, h_1, \overline{\mathcal{T}_1}, \langle \sigma_1, e_1\rangle), (\kappa_2, h_2, \overline{\mathcal{T}_2}) \xrightarrow{\pi} \overline{\mathcal{N}}, (\kappa_1, h_1', \overline{\mathcal{T}_1}, \langle \sigma_1', e_1'\rangle), (\kappa_2, h_2', \overline{\mathcal{T}_2})}$$

Figure 4: Global semantics

(b) The access happens remotely—two different nodes required.

Figure 5 shows shows the rules for the point 1, where no accesses to memory, eiher in the same node or not, are made. Each rule takes a node identifier, its heap, a stack and an

$$\textbf{[SIfTrue]}\quad \frac{}{\kappa, h, \sigma, \textsf{if true then } e_1 \textsf{ else } e_2 \xrightarrow{\varepsilon} h, \sigma, e_1}$$

$$\textbf{[SIfFalse]}\quad \frac{}{\kappa, h, \sigma, \textsf{if false then } e_1 \textsf{ else } e_2 \xrightarrow{\varepsilon} h, \sigma, e_2}$$

$$\textbf{[SLet]}\quad \frac{x \text{ fresh in } \varphi \quad \varphi' = \varphi[x \mapsto v]}{\kappa, h, \sigma.\varphi, \textsf{let } x = v \textsf{ in } e \xrightarrow{\varepsilon} h, \sigma.\varphi', e}$$

$$\textbf{[SRet]}\quad \frac{}{\kappa, h, \sigma.\varphi, \textsf{return } v \xrightarrow{\varepsilon} h, \sigma, v}$$

$$\textbf{[SVar]}\quad \frac{\varphi(x) = v}{\kappa, h, \sigma.\varphi, x \xrightarrow{\varepsilon} h, \sigma.\varphi, v}$$

$$\textbf{[SFor]}\quad \frac{x \text{ fresh in } \varphi \quad e' = (\textsf{let } x = e[n_1/i] \textsf{in for } i \textsf{ in } (n_1+1)..n_2 \textsf{ do } e)}{\kappa, h, \sigma.\varphi, \textsf{for } i \textsf{ in } n_1..n_2 \textsf{ do } e \xrightarrow{\varepsilon} h, \sigma.\varphi, e'}$$

$$\textbf{[SForSkip]}\quad \frac{n_2 > n_1}{\kappa, h, \sigma, \textsf{for } i \textsf{ in } n_1..n_2 \textsf{ do } e \xrightarrow{\varepsilon} h, \sigma, \textsf{skip}}$$

$$\textbf{[SSkip]}\quad \frac{}{\kappa, h, \sigma, \textsf{skip} \xrightarrow{\varepsilon} h, \sigma, \textsf{null}}$$

$$\textbf{[SCallL]}\quad \frac{\text{owners}(h, \alpha) = C\langle\overline{\kappa}\rangle \quad \varphi = \alpha \cdot (\textsf{this} \mapsto \alpha, x \mapsto v)}{\kappa, h, \sigma, \alpha.m(v) \xrightarrow{\varepsilon} h, \sigma.\varphi, \textsf{return } \mathcal{M}(C, m)\downarrow_3 [\overline{\kappa}]}$$

$$\textbf{[SReceiveL]}\quad \frac{\alpha\downarrow_1 = \kappa \quad h(\alpha) = (C, \overline{\kappa}, \_, m(v) :: Q) \quad e = \mathcal{M}(C, m)[\overline{\kappa}]}{\kappa, h, \alpha \cdot \emptyset, \textsf{null} \xrightarrow{\varepsilon} h[\alpha \mapsto Q], \alpha \cdot (\textsf{this} \mapsto \alpha, x \mapsto v), \textsf{return } e}$$

$$\textbf{[SContextNPE]}\quad \frac{}{\kappa, h, \sigma, E[\textsf{NPE}] \xrightarrow{\varepsilon} h, \sigma, \textsf{NPE}}$$

$$\textbf{[SNPE]}\quad \frac{}{\kappa, h, \sigma, e_{npe} \xrightarrow{\varepsilon} h, \sigma, \textsf{NPE}}$$

where e_{npe} can be $\textsf{null}.f$, $\textsf{null}.f = e$, $\textsf{null}.m(e)$, $\textsf{null}!m(e)$, $\textsf{null}[i]$, $\textsf{null}[i] = e'$

Figure 5: Semantic rules for expressions that do not perform remote operations. Null-pointer exceptions included.

expression, and reduces to a new heap, a new stack and a new expression. They have the form $\kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e'$. These rules show reduction without any communication among different nodes (they show reduction through ε). The intuition behind them is standart and similar can be found in the literature. Note only the rule for the message receiving, [**SReceiveL**], which takes an empty stack and a null expression, meaning that the expression of the thread being executed is fully reduced, and returns a new frame and expression after taking the next message in the queue to be processed. The expression returned is the body of the method asynchronously invoked, as the new frame has the values passed as arguments.

Figure 6 shows the semantic rules for the point 2. The rules on the left belong to 2(a); they have the same form of the rules introduced in Figure 5. The rules on the right belong to 2(b); they take two node identifiers, their heaps, a stack and an expression, and reduce to two new heaps, a new stack and a new expression. They have the form $\kappa_1, h_1, \sigma, e \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', e' \parallel h'_2$. In both cases they reduce through an operation described by π—the remote operation made or empty, ε (in the case of the absence of a remote operation). For instance,

[SMsgL]
$$\frac{h' = h[\langle\kappa.n\rangle :: m(v)]}{\kappa, h, \sigma, \langle\kappa.n\rangle!m(v) \xrightarrow{\varepsilon} h', \sigma, \mathsf{null}}$$

[SMsgR]
$$\frac{\pi = \mathsf{msg}(\kappa_1, \kappa_2, m) \quad h'_2 = h_2[\langle\kappa_2.n\rangle :: m(v)]}{\kappa_1, h_1, \sigma, \langle\kappa_2.n\rangle!m(v) \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, \mathsf{null} \parallel h'_2}$$

[SFReadL]
$$\frac{}{\kappa, h, \sigma, \langle\kappa.n\rangle.f \xrightarrow{\varepsilon} h, \sigma, h(\kappa.n)(f)}$$

[SFReadR]
$$\frac{\pi = \mathsf{rd}(\kappa_1, \kappa_2) \quad v = h_2(\langle\kappa_2.n\rangle)(f)}{\kappa_1, h_1, \sigma, \langle\kappa_2.n\rangle.f \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, v \parallel h_2}$$

[SFWriteL]
$$\frac{}{\kappa, h, \sigma, \langle\kappa.n\rangle.f = v \xrightarrow{\varepsilon} h[\langle\kappa.n\rangle, f \mapsto v], \sigma, v}$$

[SFWriteR]
$$\frac{\pi = \mathsf{wrt}(\kappa_1, \kappa_2) \quad h'_2 = h_2[\langle\kappa_2.n\rangle, f \mapsto v]}{\kappa_1, h_1, \sigma, \langle\kappa_2.n\rangle.f = v \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, v \parallel h'_2}$$

[SNewL]
$$\frac{\kappa = \mathcal{L}(L_1) \quad \langle\kappa.n\rangle \notin \mathrm{dom}(h) \\ h' = h[\langle\kappa.n\rangle \mapsto \mathrm{initObj}(C\langle\overline{L}\rangle)]}{\kappa, h, \sigma, \mathsf{new}\ C\langle\overline{L}\rangle \xrightarrow{\varepsilon} h', \sigma, \langle\kappa.n\rangle}$$

[SNewR]
$$\frac{\kappa_2 = \mathcal{L}(L_1) \quad \langle\kappa_2.n\rangle \notin \mathrm{dom}(h_2) \quad \pi = \mathsf{wrt}(\kappa_1, \kappa_2) \\ h'_2 = h_2[\langle\kappa_2.n\rangle \mapsto \mathrm{initObj}(C\langle\overline{L}\rangle)]}{\kappa_1, h_1, \sigma, \mathsf{new}\ C\langle\overline{L}\rangle \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, \langle\kappa_2.n\rangle \parallel h'_2}$$

[SContextL]
$$\frac{\kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e'}{\kappa, h, \sigma, E[e] \xrightarrow{\pi} h', \sigma', E[e']}$$

[SContextR]
$$\frac{\kappa_1, h_1, \sigma, e \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', e' \parallel h'_2}{\kappa_1, h_1, \sigma, E[e] \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', E[e'] \parallel h'_2}$$

Figure 6: Set of semantic rules described in 2. The left rules show the reduction of expressions that execute locally (a) and the right rules, expressions that interact with remote objects (b).

message sending in rule **[SMsgL]** adds a message to the queue of an actor in the same node as this, while **[SMsgR]** adds the message to the queue of an object in a different node. In the first case π is empty and in the second case it is $\mathsf{msg}(\kappa_1, \kappa_2, m)$. In both cases, the stack remains unchanged and the returned expression is null; namely execution is asynchronous. All the other rules, except the context rules, on the left show, as expected, reads and writes to the local heap and on the right present reads and writes to a remote heap.

4 Type Checking

Figure 7 shows the typing rules of $\mathcal{L}_{numa}$. They have the form $\overline{\Gamma} \vdash e \triangleright T, b$ where an expression e is verified against a sequence of typing contexts $\overline{\Gamma}$ resulting in a type T and an effect b. A typing context is a mapping from variables and addresses to types:

$$\Gamma \in \mathit{TypingContext} = (\mathit{var}\ \cup\ \mathit{Addr}) \rightarrow \mathit{Type}$$

The effect b describes the behaviour of e, that is, the memory accesses and messages sent to remote locations. Effects are concatenated via the function $\circ$ as defined below.

$$\varepsilon \circ b = b \qquad (bop.b_1) \circ b_2 = bop.(b_1 \circ b_2) \qquad [b_1, b_2] \circ b_3 = [b_1 \circ b_3, b_2]$$

The type T associated to an expression is found in a standard way: similar can be found

[T-Var/Addr]
$$\frac{}{\overline{\Gamma}.\Gamma \vdash var \triangleright \Gamma(var), \varepsilon \quad \overline{\Gamma}.\Gamma \vdash \alpha \triangleright \Gamma(\alpha), \varepsilon}$$

[T-True/False]
$$\frac{}{\overline{\Gamma} \vdash \mathsf{true} \triangleright \mathsf{bool}, \varepsilon \quad \overline{\Gamma} \vdash \mathsf{false} \triangleright \mathsf{bool}, \varepsilon}$$

[T-Skip/Null]
$$\frac{}{\overline{\Gamma} \vdash \mathsf{skip} \triangleright \mathsf{nil}, \varepsilon \quad \overline{\Gamma} \vdash \mathsf{null} \triangleright \mathsf{nil}, \varepsilon}$$

[T-Let]
$$\frac{\overline{\Gamma}.\Gamma \vdash e_1 \triangleright T_1, b_1 \quad x \notin \mathrm{dom}(\Gamma) \quad \overline{\Gamma}.\Gamma[x \mapsto T_1] \vdash e_2 \triangleright T_2, b_2}{\overline{\Gamma}.\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleright T_2, b_1 \circ b_2}$$

[T-Cond]
$$\frac{\overline{\Gamma} \vdash e_1 \triangleright \mathsf{bool}, b_1 \quad \overline{\Gamma} \vdash e_2 \triangleright T, b_2 \quad \overline{\Gamma} \vdash e_3 \triangleright T, b_3}{\overline{\Gamma} \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \triangleright T, b_1 \circ \{b_2\ \mathsf{or}\ b_3\}}$$

[T-For]
$$\frac{k > j \quad \overline{\Gamma} = \overline{\Gamma}'.\Gamma \quad \overline{\Gamma}'.\Gamma[i \mapsto \mathsf{int}] \vdash e \triangleright T, b}{\overline{\Gamma} \vdash \mathsf{for}\ i\ \mathsf{in}\ j..k\ \mathsf{do}\ e \triangleright T, \mathsf{Loop}(k - j + 1 \colon b)}$$

[T-Ret]
$$\frac{\overline{\Gamma} \vdash e \triangleright T, b}{\overline{\Gamma}.\Gamma \vdash \mathsf{return}\ e \triangleright T, b}$$

[T-NewO]
$$\frac{\mathrm{isActive}(C) \implies \mathrm{isMain}(\overline{\Gamma}, \mathsf{this}) \quad ot = C\langle l_1, ..., l_n\rangle \quad l_1 \neq ... \neq l_n}{\overline{\Gamma} \vdash \mathsf{new}\ ot \triangleright ot, \mathsf{wrt}(\ell(\overline{\Gamma}), l_1)}$$

[T-FWrite]
$$\frac{\overline{\Gamma} \vdash e \triangleright C\langle\overline{l}\rangle, b_1 \quad \mathcal{F}(C, f)[\overline{l}] = T \quad \overline{\Gamma} \vdash e' \triangleright T, b_2}{\overline{\Gamma} \vdash e.f = e' \triangleright T, b_1 \circ b_2 \circ \mathsf{wrt}(\ell(\overline{\Gamma}), l_1)}$$

[T-FRead]
$$\frac{\overline{\Gamma} \vdash e \triangleright C\langle\overline{l}\rangle, b_1 \quad \mathcal{F}(C, f)[\overline{l}] = T}{\overline{\Gamma} \vdash e.f \triangleright T, b_1 \circ \mathsf{rd}(\ell(\overline{\Gamma}), l_1)}$$

[T-Call]
$$\frac{\overline{\Gamma} \vdash e_1 \triangleright C\langle\overline{l}\rangle, b_1 \quad \overline{\Gamma} \vdash e_2 \triangleright T', b_2 \quad \ell(\overline{\Gamma}) = l_1 \quad \mathcal{M}(C, m)[\overline{l}] = (T, T', e_3, b_3)}{\overline{\Gamma} \vdash e_1.m(e_2) \triangleright T, b_1 \circ b_2 \circ b_3}$$

[T-Message]
$$\frac{\overline{\Gamma} \vdash e_1 \triangleright C\langle\overline{l}\rangle, b_1 \quad \overline{\Gamma} \vdash e_2 \triangleright T', b_2 \quad \ell(\overline{\Gamma}) = l_0 \quad \mathcal{M}(C, m)[\overline{l}] = (\mathsf{nil}, T', e_3, b)}{\overline{\Gamma} \vdash e_1!m(e_2) \triangleright \mathsf{nil}, b_1 \circ b_2 \circ \mathsf{msg}(l_0, l_1, m).[\varepsilon, b]}$$

Figure 7: Typing rules

in [4], therefore we focus only in the behaviour produced. The rules for variables and values, **[T-Var/Addr]**, **[T-True/False]**, **[T-Skip/Null]** result in empty effects, ε, because they do not represent any communication. The typing rule **[T-Let]** results in the concatenation of the behaviour of both expressions. The resulting behaviour of the rule **[T-Cond]** is the behaviour of the predicate concatenated with a choice type which describes the behaviour of both branches. The rule **[T-For]** returns a loop type $\mathsf{Loop}(n \colon b)$, where n is the number of iterations of the loop and b is the behavioural type of its body.

The behaviour of the creation of an object, with **[T-NewO]**, is a write behaviour, from the location of this to the location of the new object, as new data is written to memory. The predicate isActive(C) is true if the class of the object being created in annotated as active and the predicate isMain($\overline{\Gamma}$, this) is true if the class being verified is the main class. The field write is also represented by the write behaviour, given that it changes data already in memory. Typing the expression $e.f = e'$ with the rule **[T-FWrite]** returns the concatenation of the behaviour of e, the behaviour of e' and the write from the location of this to the location of the object changed. Following the same idea, the field read, $e.f$, is represented by the read behaviour and therefore the rule **[T-FRead]** gives the concatenation of the behaviour of e with a read type from the location of this and to the location of the object read. The typing rule, **[T-Call]**, describes synchronous method invocation which is only allowed if the receiver is in the same location as the this object. Its behaviour is the behaviour of the receiver concatenated with the behaviour of the expression passed as argument and the behavioural type annotated in the body of the invoked method. The typing rule for the message send, **[T-Message]**, is similar. However, it is possible to send a message to a different location and moreover it introduces parallelism in our types: the receiving of the message should be executed in parallel with the continuation of the message sending—the resulting behaviour has the continuation type, which in this case is ε, in parallel with the expression to be executed due the message received.

5 The global behaviour

We define a global behaviour, Σ, as a sequence of behavioural types

$$\Sigma \in \overline{Behaviour}$$

The behaviour of a node $\mathcal{N}$ describes the remote reads, writes and message sends to be executed by the node; it is obtained from the behaviour of the execution threads and message queues of all actors in that node. The global behaviour of a runtime configuration, $\overline{\mathcal{N}}$, describes the remote reads, writes and message sends to be executed by all nodes; it is the parallel combination of the behaviours of each the nodes $\mathcal{N}_i$. Both definitions, the behaviour of a node and the global behaviour of a configuration, are below.

Definition 1 (The global behaviour)**.**

(1) $\mathcal{N}_1, \dots, \mathcal{N}_n \blacktriangleright \overline{b}_1, \dots, \overline{b}_n$ ***iff*** $\forall i \in 1..n : \mathcal{N}_i \blacktriangleright \overline{b}_i$

(2) $\kappa, h, \langle \sigma_1, e_1 \rangle, \dots, \langle \sigma_n, e_n \rangle \blacktriangleright b_1, \dots, b_n$ ***iff*** $\forall i \in 1..n : h, \sigma_i, e_i \blacktriangleright b_i$

(3) $h, \sigma, e \blacktriangleright \text{filter}(b \circ b_1 \circ ... \circ b_n)$ ***iff*** $\exists T : h, \sigma \vdash e \triangleright T, b \wedge$
$(h(\sigma{\downarrow_1}) = (C, \kappa^+, \_, m_1(v_1) :: ... :: m_n(v_n) :: \emptyset) \;\wedge\; \forall j \in 1..n : \exists T_i : h, \sigma \vdash \mathcal{M}(C, n)[\kappa^+] \triangleright T_i, b_i)$

Using this notion of global behaviour, we implicitly assume a well-formed program and we state soundness of our typing, which says that if a well-formed configuration, $\overline{\mathcal{N}}$, with a global behaviour Σ, reduces to another configuration $\overline{\mathcal{N}}'$ through a communication step π then the resulting configuration $\overline{\mathcal{N}}'$ will have behaviour Σ' which is a reduction of Σ through π.

Theorem 1. *If* $\vdash \overline{\mathcal{N}} \;\wedge\; \overline{\mathcal{N}} \blacktriangleright \Sigma \;\wedge\; \overline{\mathcal{N}} \xrightarrow{\pi} \overline{\mathcal{N}}'$ *then* $\exists \Sigma' : \overline{\mathcal{N}}' \blacktriangleright \Sigma' \;\wedge\; \Sigma \sqsubseteq_\pi \Sigma'$

The definitions of well-formed configuration (including well-formed heap and well-formed stack) and (global) behaviour reduction are defined below:

Definition 2 (Well-formed (1) configuration, (2) node, (3) heap, (4) stack and (5) stack frame)**.**

(1) $\vdash \overline{\mathcal{N}}$ ***iff*** $\forall i, j : \mathcal{N}_i{\downarrow_1} = \mathcal{N}_j{\downarrow_1} \implies i = j \;\wedge\; \forall \mathcal{N}' : \overline{\mathcal{N}} \vdash \mathcal{N}'$

(2) $\overline{\mathcal{N}} \vdash \kappa, h, (\langle \sigma_1, e_1 \rangle, ..., \langle \sigma_n, e_n \rangle)$ ***iff***
$\forall \alpha \in \text{dom}(h) : \alpha{\downarrow_1} = \kappa \;\wedge\; h(\alpha){\downarrow_2} = \kappa, \_ \;\wedge\; \overline{\mathcal{N}} \vdash h$
$\wedge\; \forall i \in \{1..n\} : \text{heaps}(\overline{\mathcal{N}}) \vdash \sigma_i \;\wedge\; \exists T_i, b_i : h, \sigma_i \vdash e_i \triangleright T_i, b_i$

(3) $\overline{\mathcal{N}} \vdash h$ ***iff*** $\forall \alpha \in \text{dom}(h) : \text{heaps}(\overline{\mathcal{N}}) \vdash \alpha : \text{owners}(h, \alpha)$

(4) $h \vdash \alpha \cdot \varphi_1, ..., \varphi_n$ ***iff*** $\forall i \in \{1..n\} : h \vdash \varphi_i$

(5) $h \vdash (\mathsf{this} \mapsto \alpha, x_1 \mapsto v_1, \dots, x_n \mapsto v_n)$ ***iff*** $\{\alpha, v_1 ... v_n\} \subseteq \{\mathsf{true}, \mathsf{false}, \mathsf{null}\} \cup \text{dom}(h)$

Definition 3 (Global behaviour reduction)**.**

$$\Sigma \sqsubseteq_\pi \Sigma' \textbf{ iff } \Sigma = \overline{b}_1, b, \overline{b}_2 \;\wedge\; \Sigma' = \overline{b}'_1, b', \overline{b}'_2 \;\wedge\; b \sqsubseteq_\pi b' \;\wedge\; (b = [b_1, b_2] \implies b' = b_1 \;\wedge\; \exists b_j \in \Sigma, b'_j \in \Sigma' : b'_j = b_j \circ b_2)$$

Definition 4 (Behaviour reduction)**.**

$$b_1 \sqsubseteq_\pi b_2 \textbf{ iff } b_1 = \pi.b_2$$
$$b_1 \sqsubseteq_\varepsilon b_2 \textbf{ iff } b_1 = b_2 \;\vee\; b_1 = \{b_2 \text{ or } \_\} \;\vee\; b_1 = \{\_ \text{ or } b_2\} \;\vee\; (b_1 = \mathsf{Loop}(n\!: b).b' \;\wedge\; b_2 = b.\mathsf{Loop}(n-1\!: b).b') \;\vee\; b_1 = [b_2, \_]$$

Theorem 1 is a corollary of Lemmas 1 and 2.

Lemma 1. *If* $\overline{\mathcal{N}} \vdash h \ \wedge\ h \vdash \sigma \ \wedge\ \kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e' \ \wedge\ h, \sigma \vdash e \triangleright T, b \ \wedge\ \neg(\sigma{\downarrow_2}= \emptyset \ \wedge\ e = \mathsf{null})$ *then* $\exists b' : h', \sigma' \vdash e' \triangleright T, b' \ \wedge\ \text{filter}(b) \sqsubseteq_\pi \text{filter}(b')$

Lemma 2. *If* $\overline{\mathcal{N}} \vdash h_1 \ \wedge\ \overline{\mathcal{N}} \vdash h_2 \ \wedge\ h_1 \cup h_2 \vdash \sigma \ \wedge\ \kappa_1, h_1, \sigma, e \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1', \sigma', e' \parallel h_2' \ \wedge\ h_1 \cup h_2, \sigma \vdash e \triangleright T, b$ *then* $\exists b' : h_1' \cup h_2', \sigma' \vdash e' \triangleright T, b' \ \wedge\ \text{filter}(b) \sqsubseteq_\pi \text{filter}(b')$

6 Final Remarks

Related Work. To the best of our knowledge there is no integration of behavioural types in the active/passive object paradigm, or any formalism that combines behavioural types with ownership types to describe memory accesses; however there are already a few programming languages that use session (behavioural) types in actor-based languages, namely: the integration of session types in a Featherweight Erlang introduced by Mostrous and Vasconcelos [10]; an implementation of multiparty session types in an actor library written in Python presented by Neykova and Yoshida [11]; and the behavioural type system for an actor calulus, proposed by Crafa [7].

With respect to programming languages with the notion of locations and proximity among processes and data, Rinard presented an extension of the programming language Jade (an implicitly parallel programming language designed to explore task-level concurrency [16]) that allows the execution of tasks close to the data that they will use [15]. The language has constructs to describe how the processes access to the data; this information is analysed and used to improve the communication. Given that it is more expensive to access data remotely than locally, the author introduces a locality optimization algorithm that schedules the execution of tasks on places (processors) close to the data. The programming language X10 [17], developed by IBM, also features a notion of locality/places. In X10, each object can be either assigned to a place or distributed among different places. Chandra et al. [3] presented a new dependent type system for X10 that captures information about the locality of the resources in a partitioned heap for distributed data structures, called *place types*. It provides information not only about whether a reference is local or remote, but also if two remote references point to resources in the same place or not. Therefore, the compiler may use this information to decrease the runtime overhead.

Conclusion. This paper presents the fomalisation of a small oject-oriented programming language that amalgamates behavioural types with ownership types in order to describe remote memory accesses in NUMA systems. Ownership types play a role in the representation of the topology and behavioural types in the definition of reads, writes and messages sent to remote locations. This sequence of memory access operations are annotated in the method declarations as the ownership/location parameters are annotated in class declarations. This formalisation is just the first step towards a programming language that optimises performance by moving objects to nodes where they have a cheaper cost (the cost of interacting with other objects and of doing remote accesses).

Acknowledgements. This work was funded by the EU project UpScale FP7-612985 (`http://www.upscale-project.eu/`).

References

[1] Amd bulldozer server. `http://www.amd.com/en-us/products/server`.

[2] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *CONCUR 2009*, volume 5710 of *LNCS*, pages 211–228. Springer, 2009.

[3] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *PPoPP '08*, pages 11–22. ACM, 2008.

[4] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37:292–310, 2002.

[5] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS '08*, pages 139–154. Springer, 2008.

[6] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64. ACM, 1998.

[7] Silvia Crafa. Behavioural types for actor systems. Technical report, 2012.

[8] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposym on Programming*, volume 1381, pages 22–138, 1998.

[9] Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003.

[10] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session typing for a featherweight Erlang. In *COORDINATION 2011*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011.

[11] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION 2014*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.

[12] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP'98*, pages 158–185. Springer, 1998.

[13] Optimizing applications for numa. `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`.

[14] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, pages 376–385, 1993.

[15] Martin C. Rinard. Locality optimizations for parallel computing using data access information. *International Journal of High Speed Computing*, 9(2):161–179, 1997.

[16] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26, 1993.

[17] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: Concurrent programming for modern architectures. In *PPoPP '07*, pages 271–271. ACM, 2007.

[18] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, pages 157–171, 1986.

A Identifier Conventions and Semantics

Identifier conventions.

$$n \in \mathbb{N} \quad C \in \mathit{ClassId} \quad m \in \mathit{MethId} \quad f \in \mathit{FieldId} \quad L \in \mathit{LocId} \quad p \in \mathit{OwnershipId} \quad x, i \in \mathit{varId}$$

B Auxiliary definitions, shorthands and lookup functions

Definition 5 (Well-formed program and class)**.**

$$\vdash P \equiv \forall([\mathsf{active}]\ \mathsf{class}\ C\langle ...\rangle ... \in P) : P \vdash C \qquad P \vdash C \equiv \begin{cases} \mathcal{O}(C) = \{p_1, ..., p_n\} \ \wedge \\ \forall m : \mathcal{M}(C, m) = (T, x : T', e, b) \ \wedge \\ (\mathsf{this} \mapsto C\langle p_1, ..., p_n\rangle, x \mapsto T') \vdash e \triangleright T, b' \\ \implies b = \text{filter}(b') \end{cases}$$

Given that the effects returned during type checking do not exclude reads and writes happening in the same node, we apply a function filter(b) in order to exclude such annotations. The function is define as follows.

$$\text{filter}(\varepsilon) = \varepsilon \qquad \text{filter}([b_1, b_2]) = [\text{filter}(b_1), \text{filter}(b_2)]$$
$$\text{filter}(\pi.b) = (\text{if } \text{source}(\pi) = \text{dest}(\pi) \text{ then } \varepsilon \text{ else } \pi).\,\text{filter}(b)$$
$$\text{filter}(\{b_1 \ \mathsf{or}\ b_2\}.b_3) = (\text{if } \text{filter}(b_1) = \varepsilon \ \wedge \ \text{filter}(b_2) = \varepsilon \text{ then } \varepsilon \text{ else } \{\text{filter}(b_1) \ \mathsf{or}\ \text{filter}(b_2)\}).\,\text{filter}(b_3)$$
$$\text{filter}(\mathsf{Loop}(n\colon b).b') = (\text{if } \text{filter}(b) = \varepsilon \text{ then } \varepsilon \text{ else } \mathsf{Loop}(n\colon \text{filter}(b))).\,\text{filter}(b')$$

Note that if the expressiosns nested in for-loops or conditional expressions have behaviour ε, then the the loop or choice types are not annotated.

Definition 6 (Value agreement)**.**

$$\textbf{[WFTrue]}\ \frac{}{h \vdash \mathsf{true} : \mathsf{bool}} \qquad \textbf{[WFFalse]}\ \frac{}{h \vdash \mathsf{false} : \mathsf{bool}} \qquad \textbf{[WFNull]}\ \frac{T = \mathsf{nil} \ \vee \ \text{isValid}(T)}{h \vdash \mathsf{null} : T}$$

$$\textbf{[WFObj]}\ \frac{h(\alpha) = (C, (\overline{\kappa}), (f_i \mapsto v_i)_{i \in I}, \bullet) \qquad \forall i \in I : h \vdash v_i : \mathcal{F}(C, f_i)[\overline{\kappa}]}{h \vdash \alpha : C\langle\overline{\kappa}\rangle}$$

$$\textbf{[WFAObj]}\ \frac{\begin{array}{c} \textit{For } I \textit{ some index set} \quad h(\alpha) = (C, (\overline{\kappa}), (f_i \mapsto v_i)_{i \in I}, m_1(v_1) :: ... :: m_n(v_n) :: \emptyset) \\ \forall i \in I : h \vdash v_i : \mathcal{F}(C, f_i)[\overline{\kappa}] \quad h, \alpha \cdot (\mathsf{this} \mapsto \alpha, x \mapsto v_i) \vdash v_i \triangleright \mathcal{M}(C, m_i)\!\downarrow_2 [\overline{\kappa}], b \end{array}}{h \vdash \alpha : C\langle\overline{\kappa}\rangle}$$

Lookup functions Considering P, the globally accessible program, and the class declaration $\mathsf{class}\ C\langle p^+\rangle\{\overline{Fd}\ \overline{Md}\} \in P$:

$$\mathcal{O}(C) = \{p^+\} \qquad \mathcal{F}(C, f) = T \textbf{ iff } f : T \in \overline{Fd} \qquad \mathcal{F}_s(C) = \{\overline{Fd}\}$$
$$\mathcal{M}(C, m) = (T, T', e, b) \textbf{ iff } \mathsf{def}\ m(x : T') : T\ \mathsf{in}\ b\ \{e\} \in \overline{Md}$$
$$\mathcal{F}(C, f)[l_1, ..., l_n] = \mathcal{F}(C, f)[l_1/p_1, \ldots, l_n/p_n] \textbf{ where } \mathcal{O}(C) = \{p_1, \ldots, p_n\}$$

Operations on the heap

$$h[\alpha \mapsto o] = h' \textbf{ where } h'(\alpha) = o \ \wedge \ \forall \alpha_i \in \text{dom}(h) \setminus \{\alpha\} : \ h(\alpha_i) = h'(\alpha_i)$$
$$h[\alpha, f \mapsto v] = h' \textbf{ where } h'(\alpha) = h(\alpha)[f \mapsto v] \ \wedge \ \forall \alpha_i \in \text{dom}(h) \setminus \{\alpha\} : \ h(\alpha_i) = h'(\alpha_i)$$
$$h[\alpha :: m(v)] = h' \textbf{ where } h(\alpha) = o \ \wedge \ o\!\downarrow_4 \neq \bullet \ \wedge \ h' = h[\alpha \mapsto (o\!\downarrow_1, o\!\downarrow_2, o\!\downarrow_3, m(v) :: o\!\downarrow_4)] \ \wedge \ \forall \alpha_i \in \text{dom}(h) \setminus \{\alpha\} : \ h(\alpha_i) = h'(\alpha_i)$$
$$\text{owners}(h, \alpha) = C\langle\overline{\kappa}\rangle \textbf{ where } h(\alpha)\!\downarrow_1 = C \ \wedge \ h(\alpha)\!\downarrow_2 = \overline{\kappa}$$
$$h_1 \cup h_2 = h \textbf{ where } \forall \alpha \in \text{dom}(h) : h_1(\alpha) = h(\alpha) \ \vee \ h_2(\alpha) = h(\alpha)$$

Operations on objects

$$o(f) \equiv o{\downarrow_3}\,(f)$$

$$o[f \mapsto v] \equiv (o{\downarrow_1}, o{\downarrow_2}, (f \mapsto v, \overline{f_i \mapsto v_i}), o{\downarrow_4}) \qquad \textbf{where } o{\downarrow_3}= f \mapsto \_, \overline{f_i \mapsto v_i}$$

$$\text{initObj}(C\langle L_1, ..., L_m\rangle) \equiv \begin{cases} (C, \kappa_1, ..., \kappa_m, (f_i \mapsto \text{init}(T_i))_{i\in 1..n}, \emptyset) & \text{isActive}(C) \\ (C, \kappa_1, ..., \kappa_m, (f_i \mapsto \text{init}(T_i))_{i\in 1..n}, \bullet) & \text{otherwise} \end{cases}$$

$$\textbf{where } \mathcal{F}_s(C) = \{f_1 : T_1, \ldots, f_n : T_n\} \text{ and } \forall j \in \{1..m\} : \kappa_j = \mathcal{L}(L_j)$$

Operations on types

$$\text{init}(T) \equiv \text{if } T = \mathsf{bool} \text{ then } \mathsf{false} \text{ else } \mathsf{null} \qquad \ell(\overline{\Gamma}) = l \textbf{ iff } \overline{\Gamma} = \_\Gamma \;\wedge\; \Gamma(\mathsf{this}) = C\langle l, \_\rangle$$

Other definitions

$$e[C, \kappa_1, \ldots, \kappa_n] = e[\kappa_1/p_1, \ldots, \kappa_n/p_n] \text{ where } \mathcal{O}(C) = \{p_1, \ldots, p_n\}$$

$$\text{heaps}(\mathcal{N}_1, \ldots, \mathcal{N}_n) = h_1 \cup \cdots \cup h_n \textbf{ iff } \forall i \in \{1..n\} : \mathcal{N}_i{\downarrow_2}= h_i$$

$$h, \sigma \vdash e \triangleright T, b \textbf{ iff } \text{buildContext}(h, \sigma) \vdash e \triangleright T, b$$

$$\text{typeOf}(h, v) \equiv \text{if } v = \mathsf{true} \;\vee\; v = \mathsf{false} \text{ then } \mathsf{bool} \text{ else owners}(h, v)$$

$$\frac{\text{buildContext}(h, \varphi_1) = \Gamma_n \quad \ldots \quad \text{buildContext}(h, \varphi_n) = \Gamma_1}{\text{buildContext}(h, \alpha \cdot \varphi_1 \ldots \varphi_n)} \qquad \frac{T_{this} = \text{typeOf}(h, \alpha) \quad T_1 = \text{typeOf}(h, v_1) \;\ldots\; T_n = \text{typeOf}(h, v_n) \quad \Gamma = (\mathsf{this} \mapsto T_{this}, x_1 \mapsto T_1, \ldots, x_n \mapsto T_n)}{\text{buildContext}(h, \mathsf{this} \mapsto \alpha, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n) = \Gamma}$$

C Topology Example

Consider the following code with three class declarations: an active class C, a passive D and the class Main. An active object, instance of C, has three fields pointing to three objects in different locations of type D. The class main creates three abstract (or symbolic) locations L1, L2, L3 and the body of the main method.

```
active class C⟨p1, p2, p3⟩
  d1: D⟨p1⟩
  d2: D⟨p2⟩
  d3: D⟨p3⟩

class D⟨p⟩
```

```
class Main⟨L1, L2, L3⟩
  def main(): nil
    as b write(L1, L2). write(L1, L3) {
    let x = new C ⟨L1, L2, L3⟩ in
    let y = (x.d1 = new D⟨L1⟩) in
    let z = (x.d2 = new D⟨L2⟩) in
              x.d3 = new D⟨L3⟩
  }
```

The topology after execution of the main method is depicted in the following figure. In the abstract

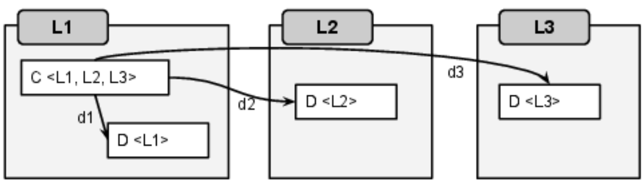

Figure 8: The ownership topology after the execution of the expression in the method main.

location L1 there is an instance of class C and an instance of class D. Abstract locations L2 and L3 have both an instance of class D. Although the programmer define 3 abstract locations, the machine might have a different number of nodes. For instance, in a system with two different nodes, we could have the mapping $(L_1 \mapsto \kappa_1, L_2 \mapsto \kappa_2, L_3 \mapsto \kappa_2)$ between abstract locations and node identifiers, which means that the objects in L1 are in the node κ_1, and objects from L2 and L3 are in the same node, as depicted in Figure 9.

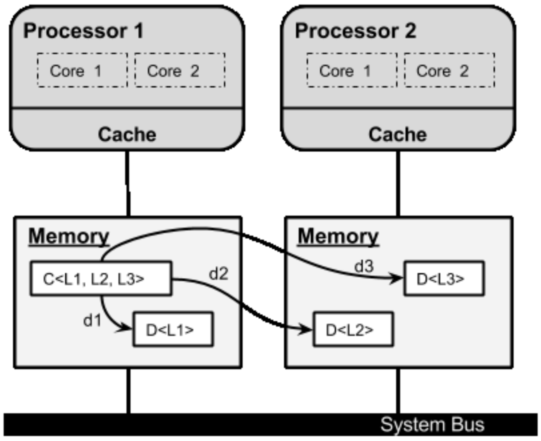

Figure 9: NUMA system with two different nodes