URL: http://www.elsevier.nl/locate/entcs/volume54.html 14 pages

Global Types and Network Services

G. Ferrari

Dip. Informatica, Univ. Pisa, Italy

E. Moggi

DISI, Univ. di Genova, Italy

R. Pugliese

Dip. di Sistemi e Informatica, Univ. di Firenze, Italy

Abstract

Mobility seems to be a fundamental aspect for global computing, however it gives rise to a lot of relevant security problems. We address the problem of protecting hosts from attacks or misbehavior of mobile processes. We propose to move process abstractions, i.e. process parameterized with respect to the operations having a local meaning, instead of processes ready-to-run or active processes (agents). Moreover, we exploit global values and types to ensure that operations having a local meaning are used only locally. Our approach is general and could be applied to every language/system for programming and coordinating network services that permits remote communications with transmission of code fragments. We illustrate our approach by using the KLAIM language, where the network services are codified by means of tuples inside network service repositories, as, e.g., in SUN JavaSpace and IBM TSpace.

1 Introduction

In global computing, process mobility seems to be a fundamental aspect, however it gives rise to a lot of relevant security problems. For instance, mobile processes threaten host machines with theft or misuse of system resources (e.g. information, money, cpu time, disk space, bandwidth). This may degrade the performance of hosts or compromise their security and reputation. Reciprocally, host machines threaten mobile processes with theft of resources (e.g. information) and reputation (e.g. the host may implant its own tasks into the

* Work supported by EC WG APPSEM and MURST projects SALADIN and TOSCA.

process for execution at other hosts) and with the possibility of killing the processes.

In this paper, we address the problem of protecting hosts from attacks or misbehavior of mobile processes. To overcome this problem, different solutions have been proposed in the literature that are based on, e.g., type systems [4,13,20,8,9], information flow analysis [12,3,14] and proof carrying code [19]. To have a fine-grain control over the behavior of mobile processes and to directly program and manage security policies, we propose to move *process abstractions*, i.e. process code which abstracts from local operations, instead of processes ready-to-run or active processes. To complete a migration, a communication between the source and the target hosts must first take place. Then, the target host can instantiate the received process abstraction, by defining all potentially dangerous operations, and can wrap the resulting process within a customized environment that takes care of process execution. In other words, depending on the trustness of the code received, the local operations over which the code is abstracted may be instantiated with implementations of local operations at the right security level thus avoiding undesired accesses to local resources.

Our approach is general and could be applied to every language/system for programming and coordinating network services that permits remote communications with transmission of code fragments. However, to put it in a concrete form, in this paper we will apply our approach to a specific language. We will consider the language Klaim [1] (Kernel Language for Agents Interaction and Mobility), an experimental kernel language, inspired by the Linda coordination model [11,5], specifically designed to model and to program distributed concurrent applications which exploit code mobility. Hence, following the Klaim approach, we assume that network services are codified by means of tuples inside network service repositories, i.e. multiple distributed tuple spaces. We remark that the Jini [1] infrastructure offers similar mechanisms (JavaSpace [21]) for matching client requests and services.

The Klaim programming paradigm identifies *processes* as the primary units of computation, and *nets*, i.e. collections of *nodes*, as the coordinators of process activities. Each node has an address, called *locality*, and consists of a process component and a tuple space component. Processes are distributed over nodes and asynchronously communicate via tuple spaces. Klaim processes may perform three different kinds of basic operations: accessing (possibly remote) tuple spaces, sending (possibly remotely) processes for execution and creating new nodes. In particular, mobility can be performed in two different ways. The (asynchronous) migration primitive **eval** allows a process to autonomously move to another node that has no control over the incoming mobile process. In addition, processes can be exchanged in communications by using the primitives **out** and **in**/**read**. This fosters synchronous mobil-

[1] The requirements and the design philosophy of the language are presented in [7]; Klaim prototype implementation is described in [2].

ity because the receiving process has a certain amount of control over the execution of the incoming process.

In this paper, we introduce HotKlaim (for Higher-order typed Klaim), a variant of Klaim that does not provide the **eval** primitive while provides a kind of data which can embody a process, namely *process abstraction.* Process abstraction is useful for parameterization with respect to local operations. To express process abstractions in HotKlaim the full power of system F [10] is needed, while $\Box$-types, similar to those in $\lambda^{\Box}$ [6], are used to classify global values that can freely move over the network.

We present a simplified version of HotKlaim, where pattern matching is replaced by lambda-abstraction and there is only one operation to read values from tuple spaces, which suffices to illustrate the key idea of the paper. However, in informal examples we make use of pattern matching and other constructs typical of functional languages.

The rest of the paper is organized as follows: Section 2 introduces the syntax of HotKlaim, Section 3 gives the type system, Section 4 defines the operational semantics, Section 5 presents the type safety result (proofs are omitted), and Section 6 illustrates a simple example of mobile code application in HotKlaim. A few comparisons with related work are in Section 7.

2 HotKlaim

Throughout this paper we will use the following notations and conventions.

- m, n range over the set N of natural numbers, and $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} | i < m\}$ of its predecessors;
- $\overline{e}$ denotes a sequence of terms and $|\overline{e}|$ indicates the number of its elements;
- if Γ_1 and Γ_2 are sequences of declarations, then Γ_1, Γ_2 denotes their concatenation (and similarly for sequences of terms);
- $\mu(A)$ is the set of multisets with elements in A, and $\uplus$ is multiset union;
- $e_2[x{:}= e_1]$ denotes substitution of e_1 for x in e_2 (modulo alpha-conversion).

HotKlaim is a kernel functional language based on system F [10], extended with the modal types of $\lambda^{\Box}$ [6] and Klaim's primitives for concurrency, communication and dynamic creation of new sites. The syntax uses a few auxiliary syntactic categories:

- a numerable set XT of *type variables*, ranged over by $X, Y, \ldots$;
- a numerable set X of *term variables*, ranged over by $x, y, \ldots$;
- a numerable set L of *localities*, ranged over by $l, l_1, \ldots$;
- a finite set $\mathsf{O} = \{nil, spawn, new, input, output\}$ of *local operations*, ranged over by o.

Local operations take a fixed number of type- and term-operands

- Types $t \in \mathsf{T}::= X \mid L \mid P \mid t_1 \to t_2 \mid (t_i|i \in m) \mid \Box t \mid \forall X{:}\, k.t$ with $k = \mathsf{g}, \mathsf{l}$

  Fully global types $g \in \mathsf{G}::= L \mid (g_i|i \in m) \mid \Box t$
- Terms $e \in \mathsf{E}::= x \mid l \mid \lambda x{:}\, t{:}\, k.e \mid e_1\ e_2 \mid \mathsf{fix}\ x{:}\, t{:}\, k.e \mid (e_i|i \in m) \mid \pi_i(e)$

  $\mid\ \Lambda X{:}\, k.e \mid e[t] \mid nil \mid o[\overline{t}](\overline{e})$

  where $|\overline{t}| = \#^t o$ and $|\overline{e}| = \#o > 0$
- Signatures $\Sigma \subseteq_{fin} \mathsf{L}$
- Contexts $\Delta, \Gamma \in \mathsf{Ctx}::= \emptyset \mid \Gamma, X{:}\, k \mid \Gamma, x{:}\, t{:}\, k$

Fig. 1. Syntax for types and terms.

- process: nil, $spawn(e)$
- new locality: $new(e)$
- communication: $input[t](e_1, e_2)$, $output[t](e_1, e_2)$

$\#^t o$ and $\#o$ will indicate the type- and term-arity of o. The local operations *input* and *output* operate over tuple spaces, *new* creates a new site, *spawn* (locally) activates a new thread of execution and, finally, *nil* stands for the terminated process.

The syntax of HotKlaim is given in Figure 1. The definition of the set FV(_) of free variables is as expected, e.g. $\mathrm{FV}(\lambda x{:}\, t{:}\, k.e) = \mathrm{FV}(t) \cup (\mathrm{FV}(e) - \{x\})$.

In HotKlaim there are two kinds of types (and terms), global- and local-types (terms), classified by g and l respectively. Contexts declare the kind of type- and term-variables. L is the global-type of localities, and P is the local-type of processes. A global-type does not contain P nor free type variables declared local. A global-term, in addition, does not contain local operations o, nor free term variables declared local.

3 Type system

The type system derives judgments of the following forms

- $\Sigma; \Gamma \vdash$, i.e. Γ is a well-formed context
- $\Sigma; \Gamma \vdash t$, i.e. t is a well-formed type
- $\Sigma; \Gamma \vdash e{:}\, t$, i.e. e is a well-formed term of type t.

We will use $\Sigma; \Gamma \vdash J$ to indicate any of the judgements above. Moreover, we will use Γ^{g} to indicate the context Γ where local declarations, i.e. those of the form $X{:}\, \mathsf{l}$ and $x{:}\, t{:}\, \mathsf{l}$, have been removed. Finally, we will use also the derived judgement $\Sigma; \Gamma^{\mathsf{g}} \vdash_{\mathsf{g}} J$ to stand for $\Sigma; \Gamma^{\mathsf{g}} \vdash J$ with the additional requirement that P and the local operation o are not used in J.

The typing rules are given in Figure 2 (in some rules, when clear from the context, the premise $\Sigma;\Gamma\vdash$ is removed). Most of the rules are standard. The typing rules for the local operations o are simpler to understand by viewing o as a higher-order polymorphic constants:

- $nil\colon t_{nil}[P] \stackrel{\Delta}{=} P$
- $spawn\colon t_{spawn}[P] \stackrel{\Delta}{=} (() \to P) \to ()$
- $new\colon t_{new}[P] \stackrel{\Delta}{=} (L \to P) \to L$
- $input\colon t_{input}[P] \stackrel{\Delta}{=} \forall X\colon \mathsf{g}.(L, \Box X \to P) \to P$
- $output\colon t_{output}[P] \stackrel{\Delta}{=} \forall X\colon \mathsf{g}.(L, \Box X) \to ()$

For instance, the polymorphic type $\forall X\colon \mathsf{g}.(L, \Box X \to P) \to P$ abstracts over the type X of global values (which can be read from tuple spaces), and states that *input* takes as parameters a locality (of type L) and a continuation function (of type $\Box X \to P$) that given a global value yields the continuation process. Hence, the *input* operation returns the continuation process.

We can now comment on the typing rules in Figure 2. The bound variable in $\lambda x\colon t\colon k.e$ and $\mathsf{fix}\ x\colon t\colon k.e$ can be declared either local or global, depending on whether k is l or g. When $k=\mathsf{l}$ the typing rules are the standard ones.

The formation rule for $\Box t$ requires t to be a global-type. The introduction and elimination rules for $\Box t$ say that $\Box t$ is a subset of t (i.e. it classifies a subset of the terms classified by t), and that $\Box t$ is equal to t (i.e. $\Box t$ and t classify the same set of terms) when $t \in \mathsf{G}$. A key property of $\Box t$ is that its values are exactly the global values of type t.

Only values of type $\Box t$ can be exchanged with the *input* and *output* operations. In particular, one cannot exchange processes. However, one can exchange global values of type $Code \stackrel{\Delta}{=} \forall X\colon \mathsf{l}.(t_o[X] | o \in \mathsf{O}) \to X$. Intuitively, $Code$ is the type of *process code* abstracted over the local type P and the local operations in O. Its definition mimics that given in [17,18] for *monadic code*. When a global value of type $Code$ has been read, it can be turned into a process by applying it to the local type P and the tuple $(o|o \in \mathsf{O})$ of local operations.

Lemma 3.1 (Substitution) *The following rules are admissible (cf [6]):*

$$\frac{\Sigma;\Gamma_1^{\mathsf{g}} \vdash_{\mathsf{g}} t \quad \Sigma;\Gamma_1, X\colon \mathsf{g}, \Gamma_2 \vdash J}{\Sigma;\Gamma_1, \Gamma_2[X\colon= t] \vdash J[X\colon= t]} \qquad \frac{\Sigma;\Gamma_1^{\mathsf{g}} \vdash_{\mathsf{g}} e\colon t \quad \Sigma;\Gamma_1, x\colon t\colon \mathsf{g}, \Gamma_2 \vdash J}{\Sigma;\Gamma_1, \Gamma_2 \vdash J[x\colon= e]}$$

$$\frac{\Sigma;\Gamma_1 \vdash t \quad \Sigma;\Gamma_1, X\colon \mathsf{l}, \Gamma_2 \vdash J}{\Sigma;\Gamma_1, \Gamma_2[X\colon= t] \vdash J[X\colon= t]} \qquad \frac{\Sigma;\Gamma_1 \vdash e\colon t \quad \Sigma;\Gamma_1, x\colon t\colon \mathsf{l}, \Gamma_2 \vdash J}{\Sigma;\Gamma_1, \Gamma_2 \vdash J[x\colon= e]}$$

$$\frac{\Sigma;\Gamma\vdash}{\Sigma;\Gamma,X{:}\,k\vdash}\;X\text{ fresh}\qquad\frac{\Sigma;\Gamma\vdash t}{\Sigma;\Gamma,x{:}\,t{:}\,\mathsf{l}\vdash}\;x\text{ fresh}\qquad\frac{\Sigma;\Gamma^{\mathsf{g}}\vdash_{\mathsf{g}} t}{\Sigma;\Gamma,x{:}\,t{:}\,\mathsf{g}\vdash}\;x\text{ fresh}$$

$$\frac{\Sigma;\Gamma\vdash t_1\quad\Sigma;\Gamma\vdash t_2}{\Sigma;\Gamma\vdash t_1\to t_2}\qquad\frac{\Sigma;\Gamma\vdash t_i\quad i\in m}{\Sigma;\Gamma\vdash(t_i|i\in m)}\qquad\frac{\Sigma;\Gamma^{\mathsf{g}}\vdash_{\mathsf{g}} t}{\Sigma;\Gamma\vdash\Box t}\qquad\frac{\Sigma;\Gamma,X{:}\,k\vdash t}{\Sigma;\Gamma\vdash\forall X{:}\,k.t}$$

$$\frac{\Sigma;\Gamma\vdash}{\Sigma;\Gamma\vdash L}\qquad\frac{\Sigma;\Gamma\vdash}{\Sigma;\Gamma\vdash P}\qquad\frac{\Sigma;\Gamma\vdash}{\Sigma;\Gamma\vdash l{:}\,L}\;l\in\Sigma\qquad\frac{\Sigma;\Gamma\vdash}{\Sigma;\Gamma\vdash x{:}\,t}\;x{:}\,t{:}\,k\in\Gamma$$

$$\frac{\Sigma;\Gamma\vdash}{\Sigma;\Gamma\vdash nil{:}\,P}\qquad\frac{\Sigma;\Gamma\vdash e{:}\,()\to P}{\Sigma;\Gamma\vdash spawn(e){:}\,()}\qquad\frac{\Sigma;\Gamma\vdash e{:}\,L\to P}{\Sigma;\Gamma\vdash new(e){:}\,L}$$

$$\frac{\Sigma;\Gamma^{\mathsf{g}}\vdash_{\mathsf{g}} t\quad\Sigma;\Gamma\vdash e_1{:}\,L,\;e_2{:}\,\Box t\to P}{\Sigma;\Gamma\vdash input[t](e_1,e_2){:}\,P}\qquad\frac{\Sigma;\Gamma^{\mathsf{g}}\vdash_{\mathsf{g}} t\quad\Sigma;\Gamma\vdash e_1{:}\,L,\;e_2{:}\,\Box t}{\Sigma;\Gamma\vdash output[t](e_1,e_2){:}\,()}$$

$$\frac{\Sigma;\Gamma,x{:}\,t_1{:}\,\mathsf{l}\vdash e{:}\,t_2}{\Sigma;\Gamma\vdash(\lambda x{:}\,t_1{:}\,\mathsf{l}.e){:}\,t_1\to t_2}\qquad\frac{\Sigma;\Gamma,x{:}\,t_1{:}\,\mathsf{g}\vdash e{:}\,t_2}{\Sigma;\Gamma\vdash(\lambda x{:}\,t_1{:}\,\mathsf{g}.e){:}\,\Box t_1\to t_2}$$

$$\frac{\Sigma;\Gamma\vdash e_1{:}\,t_1\to t_2\wedge e_2{:}\,t_1}{\Sigma;\Gamma\vdash e_1\;e_2{:}\,t_2}\qquad\frac{\Sigma;\Gamma,x{:}\,t{:}\,\mathsf{l}\vdash e{:}\,t}{\Sigma;\Gamma\vdash\mathsf{fix}\;x{:}\,t{:}\,\mathsf{l}.e{:}\,t}\qquad\frac{\Sigma;\Gamma^{\mathsf{g}},x{:}\,t{:}\,\mathsf{g}\vdash_{\mathsf{g}} e{:}\,t}{\Sigma;\Gamma\vdash\mathsf{fix}\;x{:}\,t{:}\,\mathsf{g}.e{:}\,\Box t}$$

$$\frac{\Sigma;\Gamma\vdash e_i{:}\,t_i\quad i\in m}{\Sigma;\Gamma\vdash(e_i|i\in m){:}\,(t_i|i\in m)}\qquad\frac{\Sigma;\Gamma\vdash e{:}\,(t_i|i\in m)}{\Sigma;\Gamma\vdash\pi_i(e){:}\,t_i}\;i\in m$$

$$\frac{\Sigma;\Gamma^{\mathsf{g}}\vdash_{\mathsf{g}} e{:}\,t}{\Sigma;\Gamma\vdash e{:}\,\Box t}\qquad\frac{\Sigma;\Gamma\vdash e{:}\,g}{\Sigma;\Gamma\vdash e{:}\,\Box g}\qquad\frac{\Sigma;\Gamma\vdash e{:}\,\Box t}{\Sigma;\Gamma\vdash e{:}\,t}\qquad\frac{\Sigma;\Gamma,X{:}\,k\vdash e{:}\,t}{\Sigma;\Gamma\vdash(\Lambda X{:}\,k.e){:}\,\forall X{:}\,k.t}$$

$$\frac{\Sigma;\Gamma\vdash e{:}\,\forall X{:}\,\mathsf{l}.t_2\quad\Sigma;\Gamma\vdash t_1}{\Sigma;\Gamma\vdash e[t_1]{:}\,t_2[X{:=}\,t_1]}\qquad\frac{\Sigma;\Gamma\vdash e{:}\,\forall X{:}\,\mathsf{g}.t_2\quad\Sigma;\Gamma^{\mathsf{g}}\vdash_{\mathsf{g}} t_1}{\Sigma;\Gamma\vdash e[t_1]{:}\,t_2[X{:=}\,t_1]}$$

Fig. 2. Type System.

4 Operational semantics

An HotKlaim net $N\in\mathsf{Net}\overset{\Delta}{=}\mu(\mathsf{L}\times\mathsf{E})$ is a multiset of pairs $(l{:}\,e)$ consisting of a locality l (node name) and a term e (either a process running under the authority of that node or a value in the tuple space of that node). The dynamics of a net is given by a transition relation $\Longrightarrow\subset\mathsf{Net}\times(\mathsf{Net}+\{\mathsf{err}\})$, written $N\Longrightarrow N'$ or $N\Longrightarrow\mathsf{err}$, defined in terms of a transition relation $\longmapsto\;\subset\;(\mathsf{Sig}\times\mathsf{E}\times\mathsf{A}\times\mathsf{E})+(\mathsf{Sig}\times\mathsf{E}\times\{\mathsf{err}\})$, written $\Sigma;e\overset{a}{\longmapsto}e'$ or

- Values $v \in \mathsf{V}\!::= l \mid \lambda x\!: t\!: k.e \mid (v_i | i \in m) \mid \Lambda X\!: k.e$
- Redexes $r \in \mathsf{R}\!::= x \mid v_1\ v_2 \mid \mathsf{fix}\ x\!: t\!: k.e \mid \pi_i(v) \mid v[t]$
  $\mid\ o[\overline{t}](\overline{v})$ where $|\overline{t}| = \#^t o$ and $|\overline{v}| = \#o > 0$
- Eval. contexts $E \in \mathsf{EC}\!::= [\,] \mid E\ e \mid v\ E \mid (\overline{v}, E, \overline{e}) \mid \pi_i(E) \mid E\ [t]$
  $\mid\ o[\overline{t}](\overline{v}, E, \overline{e})$ where $|\overline{t}| = \#^t o$ and $|\overline{v}, \overline{e}| + 1 = \#o$

Fig. 3. Values, redexes and evaluation contexts.

$\Sigma; e \longmapsto \mathsf{err}$, describing the potential interactions of a term with the rest of the net. The set A of potential interactions is

$$a \in \mathsf{A}\!::= \tau \mid i(v)@l \mid o(v)@l \mid s(e) \mid l\!: e$$

For instance, $i(v)@l$ is the capability of *inputing* a value v from the tuple space at locality l, while $l\!: e$ is the (non-blocking) action of creating a new locality l running process e.

We follow [22] and define $\longmapsto$ in terms of redexes and evaluation contexts (see Table 3), and a reduction $\longrightarrow\ \subset (\mathsf{Sig} \times \mathsf{R} \times \mathsf{A} \times \mathsf{E}) + (\mathsf{Sig} \times \mathsf{R} \times \{\mathsf{err}\})$ for redexes given in Figure 4.

The reduction makes use of the signature to check whether there is a node with name l in the net. The reduction for $\mathit{input}[t](l, v_2)$ corresponds to an *early* semantics (according to the terminology used for the π-calculus [16]). Following the Klaim type system [8,9], *input* performs run-time type checking with the additional requirement that the input value must be global.

We can now define the transition relation $\longmapsto$ as the least relation satisfying the following rules:

$$\frac{\Sigma; r \xrightarrow{a} e}{\Sigma; E[r] \overset{a}{\longmapsto} E[e]} \qquad \frac{\Sigma; r \longrightarrow \mathsf{err}}{\Sigma; E[r] \longmapsto \mathsf{err}}$$

Finally, the net transition relation $\Longrightarrow$ is given in Figure 5. As in Linda-like languages, *input* removes one matching value chosen non-deterministically (since $\uplus$ is commutative), and such value could be in the same locality of the process performing the *input* operation (and similarly for *output*).

Lemma 4.1 (Basic properties) *A value $v \in \mathsf{V}$ cannot be of the form $E[r]$. For any $e \in \mathsf{E}$ there is at most one $E \in \mathsf{EC}$ and one $r \in \mathsf{R}$ s.t. $e \equiv E[r]$, moreover* $\mathrm{FV}(r) \subseteq \mathrm{FV}(e)$.

Notice that *nil* is not a value but is inactive.

Lemma 4.2 (Replacement for Evaluation Contexts)

- $x \longrightarrow \mathsf{err}$
- $(\lambda x\colon t\colon k.e)\ v_2 \xrightarrow{\tau} e[x\colon = v_2]$ otherwise $v_1\ v_2 \longrightarrow \mathsf{err}$
- $\mathsf{fix}\ x\colon t\colon k.e \xrightarrow{\tau} e[x\colon = \mathsf{fix}\ x\colon t\colon k.e]$
- $\pi_i(v_i|i \in m) \xrightarrow{\tau} v_i$ when $i \in m$ otherwise $\pi_i(v) \longrightarrow \mathsf{err}$
- $(\Lambda X\colon k.e)[t] \xrightarrow{\tau} e[X\colon = t]$ otherwise $v[t] \longrightarrow \mathsf{err}$
- $\mathit{spawn}(v) \xrightarrow{s(v())} ()$
- $\Sigma; \mathit{new}(v) \xrightarrow{l:(v\ l)} l$ provided $l \notin \Sigma$
- $\Sigma; \mathit{input}[t](l, v_2) \xrightarrow{i(v)@l} v_2\ v$ when $l \in \Sigma$ and $\Sigma; \emptyset \vdash_{\mathsf{g}} v\colon t$

  $\Sigma; \mathit{input}[t](v_1, v_2) \longrightarrow \mathsf{err}$ when $v_1 \notin \Sigma$
- $\Sigma; \mathit{output}[t](l, v) \xrightarrow{o(v)@l} ()$ when $l \in \Sigma$

  $\Sigma; \mathit{output}[t](v_1, v) \longrightarrow \mathsf{err}$ when $v_1 \notin \Sigma$

Fig. 4. Reduction relation. Σ is left implicit when irrelevant.

$$\frac{\Sigma(N) \cup \{l\}; e \overset{\tau}{\longmapsto} e'}{N \uplus (l\colon e) \Longrightarrow N \uplus (l\colon e')} \qquad \frac{\Sigma(N) \cup \{l\}; e \longmapsto \mathsf{err}}{N \uplus (l\colon e) \Longrightarrow \mathsf{err}}$$

$$\frac{\Sigma(N) \cup \{l\}; e \overset{s(e_1)}{\longmapsto} e_2}{N \uplus (l\colon e) \Longrightarrow N \uplus (l\colon e_1) \uplus (l\colon e_2)} \qquad \frac{\Sigma(N) \cup \{l_1\}; e \overset{l_2:e_2}{\longmapsto} e_1}{N \uplus (l_1\colon e) \Longrightarrow N \uplus (l_1\colon e_1) \uplus (l_2\colon e_2)}$$

$$\frac{\Sigma(N) \cup \{l_1, l_2\}; e \overset{i(v)@l_2}{\longmapsto} e'}{N \uplus (l_1\colon e) \uplus (l_2\colon v) \Longrightarrow N \uplus (l_1\colon e') \uplus (l_2\colon \mathit{nil})}$$

$$\frac{\Sigma(N) \cup \{l_1\}; e \overset{o(v)@l_2}{\longmapsto} e'}{N \uplus (l_1\colon e) \Longrightarrow N \uplus (l_1\colon e') \uplus (l_2\colon v)}$$

Fig. 5. Net transition relation. $\Sigma(N)$ is the signature $\{l|\exists e.(l\colon e) \in N\}$.

If $\Sigma; \Gamma \vdash E[e]\colon t$ *with* $E \in \mathsf{EC}$*, then there exists* t_1 *such that*

- $\Sigma; \Gamma \vdash e\colon t_1$
- $\Sigma'; \Gamma \vdash e'\colon t_1$ *implies* $\Sigma'; \Gamma \vdash E[e']\colon t$ *for any* $\Sigma' \supseteq \Sigma$ *and* e'

5 Type Safety

To state the type safety property we define when a net is well-formed.

Definition 5.1 A net N is well-formed if, and only if, for every $(l\colon e) \in N$

either $\Sigma(N); \emptyset \vdash e\colon P$, or $e \in \mathsf{V}$ and $\Sigma(N); \emptyset \vdash e\colon g$ for some $g \in G$
(i.e. every e in N is either a local process or a global value)

where $\Sigma(N)$ is the signature $\{l | \exists e.(l\colon e) \in N\}$.

The following Lemma captures a key property of $\Box$-types.

Lemma 5.2 *If* $\Sigma; \Gamma \vdash v\colon g$*, then* $\Sigma; \Gamma^{\mathsf{g}} \vdash_{\mathsf{g}} v\colon g$.

Proof. By induction on the derivation of $\Sigma; \Gamma \vdash v\colon g$ $\Box$

The following Lemmas express type safety for $\longrightarrow$ and $\longmapsto$.

Lemma 5.3 (Safety for $\longrightarrow$) *If* $\Sigma; \emptyset \vdash r\colon t$*, then*

- $\Sigma; r \not\longrightarrow \mathsf{err}$
- $\Sigma; r \xrightarrow{\tau} e'$ *implies* $\Sigma; \emptyset \vdash e'\colon t$
- $\Sigma; r \xrightarrow{s(e_2)} e_1$ *implies* $\Sigma; \emptyset \vdash e_2\colon P$ *and* $\Sigma; \emptyset \vdash e_1\colon t$
- $\Sigma; r \xrightarrow{l:e_2} e_1$ *implies* $l \notin \Sigma$, $\Sigma, l; \emptyset \vdash e_2\colon P$ *and* $\Sigma, l; \emptyset \vdash e_1\colon t$
- $\Sigma; r \xrightarrow{i(v)@l} e'$ *implies* $l \in \Sigma$, $\Sigma; \emptyset \vdash v\colon g$ *for some* $g \in \mathsf{G}$ *and* $\Sigma; \emptyset \vdash e'\colon t$
- $\Sigma; r \xrightarrow{o(v)@l} e'$ *implies* $l \in \Sigma$, $\Sigma; \emptyset \vdash v\colon g$ *for some* $g \in \mathsf{G}$ *and* $\Sigma; \emptyset \vdash e'\colon t$

Proof. By case analysis on the possible reductions $\Box$

Lemma 5.4 (Safety for $\longmapsto$) *If* $\Sigma; \emptyset \vdash e\colon t$*, then*

- $\Sigma; e \not\longmapsto \mathsf{err}$
- $\Sigma; e \overset{\tau}{\longmapsto} e'$ *implies* $\Sigma; \emptyset \vdash e'\colon t$
- $\Sigma; e \overset{s(e_2)}{\longmapsto} e_1$ *implies* $\Sigma; \emptyset \vdash e_2\colon P$ *and* $\Sigma; \emptyset \vdash e_1\colon t$
- $\Sigma; e \overset{l:e_2}{\longmapsto} e_1$ *implies* $l \notin \Sigma$, $\Sigma, l; \emptyset \vdash e_2\colon P$ *and* $\Sigma, l; \emptyset \vdash e_1\colon t$
- $\Sigma; e \overset{i(v)@l}{\longmapsto} e'$ *implies* $l \in \Sigma$, $\Sigma; \emptyset \vdash v\colon g$ *for some* $g \in \mathsf{G}$ *and* $\Sigma; \emptyset \vdash e'\colon t$
- $\Sigma; e \overset{o(v)@l}{\longmapsto} e'$ *implies* $l \in \Sigma$, $\Sigma; \emptyset \vdash v\colon g$ *for some* $g \in \mathsf{G}$ *and* $\Sigma; \emptyset \vdash e'\colon t$

Proof. If e is active, then it must be of the form $E[r]$. By Lemma 4.2 there exists a t_1 s.t. $\Sigma; \emptyset \vdash r\colon t_1$, then one can exploit Safety for $\longrightarrow$ $\Box$

Theorem 5.5 (Type Safety) *Let N be a well-formed net, then*

- $N \not\Longrightarrow \mathsf{err}$

- $N \Longrightarrow N'$ *implies that* N' *is well-formed and* $\Sigma(N) \subseteq \Sigma(N')$.

Proof. If N is active, then there is a pair $(l\colon e) \in N$ s.t. e is active. Since N is well-formed, then $\Sigma(N); \emptyset \vdash e\colon P$, and one can exploit Safety for $\longmapsto$ $\square$

6 A simple example: a dynamic news-gatherer

In this section, by means of a simple example borrowed from [7], we show how to use HotKlaim to program *mobile code applications.* Let us consider the following scenario. User `U` needs additional information on a piece of data represented by `item` (e.g. `item` could be the title of a book, and `U` wants to know its price). Part of the behavior of `U` depends on this information. However, there are some activities which are independent of it. `U` can look for the required information in a database distributed over the net by using a mobile process abstraction that dynamically travels among nodes looking for a tuple that contains information on `item`. We assume that each node of the database contains either a tuple of the form `(item, d)`, with the desired information, or a tuple of the form `(item, l_n)`, with the information about the next node where the search must be repeated.

In the rest of this section, we will use Klaim syntax for patterns

$p\colon := x!t\colon k \mid e \mid (p_i | i \in m)$

and ML-like notation for functions and local declarations. Moreover, for type-setting reasons, we write $\{\_\}$ instead of $\square\_$.

Given the types of the local operations (`nil` has type `P`):

```
I[P] = VX:g.(L,{X}->P)->P;       (* type of input *)
O[P] = VX:g.(L,{X})->();         (* type of output *)
N[P] = ()->P;                    (* type of delayed nil *)
```

we will make use of the following additional global types:

```
Bool = ...;
Data = ...;              (* items and associated information *)
Key  = ...;              (* authorization keys *)
L = ...;                 (* localities *)
EnvK[X] = (L,Key->I[X],Key->O[X],N[X])
MC = VX:l. EnvK[X] -> X;         (* mobile code with keys *)
```

The example uses three auxiliary functions:

```
start : {(L,L,Data,Key)} -> P ;
gatherer : {(L,Data,Key)} -> {MC} ;
execute : (L,Key->Bool,(Key,L)->Bool) -> P ;
```

`start` is the part of the behavior of user `U` that depends on the information associated to `item`. It is a process parameterized with respect to its locality `self`, the locality `l_i` where the search must start (which can be chosen according to the search key `item`), the search key `item`, and an authorization key `k` (to remotely perform local operations). `start` adds a tuple containing the process abstraction `gatherer(self, item, k)` of type `MC` to the tuple space at `l_i`, then waits for the result of the search at the local tuple space.

```
start (self,l_i:L:g, item:Data:g, k:Key:g) =
  output [MC] (l_i, gatherer(self, item, k));
  input (self,{x!Data} => ... )
```

`gatherer` is the mobile code for searching. Its parameters are: the locality `res` where the result of the search must be placed, the search key `item`, and an authorization key `k` used to check the permission to perform local operations. Since `gatherer` is mobile code of type `MC`), it is abstracted with respect to customized versions (which take a key as parameter) of the communication operations. `gatherer` looks for one of two alternative tuples. The first one contains the wanted information associated to `item` (e.g. the price); if it is found, then the result of the search is added to the tuple space at `res` and the process terminates. The second tuple contains the address `l` of the node where the search has to be repeated; if it is found, then a tuple containing the process abstraction `gatherer(res, item, k)` is added to the tuple space at `l` for searching there.

```
gatherer (res:L:g, item:Data:g, k:Key:g) =
  fix ng:MC:g.
    Fn X:l fn (self, in', out', nil'):EnvK[X]:g
      => in' k (self, (d!Data,item) => out' k (res, d); nil'() |
                      (l!L,item)    => out' k (l,ng); nil'() )
```

`execute` is the process that acts as the *guardian* of nodes. It is parameterized with respect to its locality `self`, an authorization function `safe` that, on the basis of an authorization key, checks if code can be considered reliable (and thus may not be constrained), and an authorization function `allow`, on the basis of an authorization key and of a location, check if code can be allowed to perform a communication operation at that location. `execute` takes care of taking a process abstraction (i.e. a tuple with just one field of type `MC`) from the local tuple space, specializing and instantiating the abstraction and then executing the resulting process. `in'` and `out'` are the customized versions of the communication operations with which mobile code is specialized; they make use of the two authorization functions `safe` and `allow`.

```
execute (self:L:l, safe:Key->Bool:l, allow:(Key,L)->Bool:l) =
    fix exec:P:l.
      let fun in' (k:Key) =
                if safe k then input
                else Fn [X:g] fn (l:L, f:{X}->P) =>
                     if allow(k,l) then input[X](l,f) else nil ;
          fun out' (k:Key) =
                if safe k then output
                else Fn [X:g] fn (l:L, x:{X}) =>
                     if allow(k,l) then output[X](l,x) else () ;
          fun nil' () = nil
      in
        input (self, mc!{MC} => spawn(mc[P](self,in',out',nil')); exec)
```

Finally, a HotKlaim net for this application could be

```
(l_U: start(l_U,l_i,item,k)) (l_U: execute(l_U,safe_U,allow_U))
(l_i: (item, l_1)) (l_i: execute(l_i,safe_i,allow_i))
(l_1: (item, l_2)) (l_1: execute(l_1,safe_1,allow_1))
(l_2: (item, data)) (l_2: execute(l_2,safe_2,allow_2))
```

the wanted information is supposed to be at `l_2`, and each locality has its own *guardian*, obtained by instantiating `execute` with appropriate parameters.

7 Related Work

The programming paradigms that are closer to ours are those in [23,24,15].

In [23,24] a process language, named $D\pi\lambda$, is considered that results from the integration of the call-by-value λ-calculus and the π-calculus, together with primitives for process distribution and remote process creation. Differently from HotKlaim, communication is channel based and processes cannot explicitly refer localities (indeed, these are anonymous). $D\pi\lambda$ permits the transmission of process abstractions parameterized with respect to resource (i.e. channel) names.

More specifically, in [23], a type system for $D\pi\lambda$ is defined that ensures that at any one time all the processes that intend to perform inputs at a given channel are co-located. To this aim, a value is deemed sendable whenever its exportation does not violate locality of channels. Although the type system has a different flavor from that of HotKlaim, sendable values and types play a role similar to our global values and types.

In [24], a fine-grain type system for $D\pi\lambda$ is defined that permits controlling the effect of transmitted process abstractions on local resources (i.e. channels). Differently from HotKlaim, processes are assigned fine-grain types that, like interfaces, record the resources to which processes have access together with the corresponding capabilities, and process abstractions are assigned dependent functional types that abstract from channel names and types. Although process abstractions have not polymorphic types as in HotKlaim, channel names may appear and be bound both in terms and in types and thus, in some sense, play the role of type variables.

In [15] a higher-order functional language, named Confined-λ, is presented that supports distributed computing by allowing expressions at different localities to communicate via channels. In Confined-λ, authors of code can assign regions (i.e. subsystems) to values in order to limit the part of a system where a value can freely move. Then, a type system is defined that guarantees that each value can roam only within the corresponding region. Differently from HotKlaim, communication is channel based, the transmissible process abstractions can be parameterized with respect channel names, and the types of transmissible values permit restricting the subsystem where a value can freely move.

Acknowledgements We thank the anonymous referees for their useful comments.

References

[1] K. Arnold, B. Osullivan, R.W. Scheifler, J. Waldo, A. Wollrath, B. O'Sullivan. *The Jini specification.* Addison Wesley, 1999.

[2] L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese. Interactive Mobile Agents in `XKlaim`. In *Proc. of 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE) '98*, pp.110-115, IEEE Computer Society Press, 1998.

[3] C. Bodei, P. Degano, F. Nielson, H.R. Nielson. Static analysis of processes for no read-up and no write-down. In *Proc. of FOSSACS'99*, *LNCS* 1578, pp.120-134, Springer, 1999.

[4] L. Cardelli, A. Gordon. Types for Mobile Ambients. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pp.79-92, ACM Press, 1999.

[5] N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, ACM Press, 1989.

[6] R. Davies, F. Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, St.Petersburg Beach, Florida, January 1996.

[7] R. De Nicola, G. L. Ferrari, R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility, *IEEE Transactions on Software Engineering*, 24(5):315-330, IEEE Computer Society, 1998.

[8] R. De Nicola, G. L. Ferrari, R. Pugliese. Types as Specifications of Access Policies. In In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects* (J. Vitek, C. Jensen, Eds.), LNCS State-Of-The-Art-Survey, *LNCS* 1603, Springer, pp.117-146, 1999.

[9] R. De Nicola, G. L. Ferrari, R. Pugliese, B. Venneri. Types for Access Control. *Theoretical Computers Science*, 240(1):215-254, special issue on Coordination, Elsevier Science, July 2000.

[10] J-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. PhD. Thesis, Université Paris VII, 1972.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, ACM Press, 1985.

[12] N. Heintze, J.G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. of the ACM Conf. on Principles of Programming Languages (POPL)*, pp.365-377, ACM Press, 1998.

[13] M. Hennessy, J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects* (J. Vitek, C. Jensen, Eds.), LNCS State-Of-The-Art-Survey, *LNCS* 1603, Springer, pp.95-115, 1999.

[14] M. Hennessy, J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proc. of Int. Colloquium on Automata, Languages and Programming (ICALP)*, *LNCS* , Springer, pp.415-427, 2000.

[15] D. Kirli. Confined mobile functions. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, IEEE Computer Society, 2001.

[16] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes, (Part I and II). *Information and Computation*, 100:1-77, 1992.

[17] E. Moggi, F. Palumbo. Monadic encapsulation of effects: A revised approach. *Electronic Notes in Theoretical Computer Science*, **26**, 119–136. Third International Workshop on Higher Order Operational Techniques in Semantics, 1999.

[18] E. Moggi, A. Sabry. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming*, to appear.

[19] G. Necula. Proof-carrying-code. *Proc. of the ACM Symposium on Principles of Programming Languages*, 1997.

[20] J. Riely, M. Hennessy. Trust and partial typing in open systems of mobile agents. *Proc. of the ACM Symposium on Principles of Programming Languages*, pp.93-104, ACM Press, 1999.

[21] Sun Microsystems. The JavaSpace Specifications. Available at `http://java.sun.com/products/javaspaces`, 1997.

[22] A.K. Wright, M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.

[23] N. Yoshida, M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proc. of the Int. Conf. on Concurrency Theory (CONCUR)*, *LNCS* 1664, Springer, pp. 557-572, 1999.

[24] N. Yoshida, M. Hennessy. Assigning types to processes. CS Technical Report 02/99, University of Sussex, 1999.
Available at: `http://www.mcs.le.ac.uk/ñyoshida`,