*URL:* [*http://www.elsevier.nl/locate/entcs/volume54.html*](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 protect- ing hosts from attacks or misbehavior of mobile processes. We propose to move process abstractions, i.e. process parameterized with respect to the operations hav- ing 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 codi ed 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, how-* ever 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. Recipro- cally, 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.*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

*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, di erent so- lutions have been proposed in the literature that are based on, e.g., type systems [4,13,20,8,9], information ow analysis [12,3,14] and proof carrying code [19]. To have a ne-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 migra- tion, a communication between the source and the target hosts must rst take place. Then, the target host can instantiate the received process abstraction, by de ning all potentially dangerous operations, and can wrap the resulting process within a customized environment that takes care of process execu- tion. 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 com- munications with transmission of code fragments. However, to put it in a concrete form, in this paper we will apply our approach to a speci c language. We will consider the language Klaim 1 (Kernel Language for Agents Inter- action and Mobility), an experimental kernel language, inspired by the Linda coordination model [11,5], speci cally designed to model and to program dis- tributed concurrent applications which exploit code mobility. Hence, following the Klaim approach, we assume that network services are codi ed by means of tuples inside network service repositories, i.e. multiple distributed tuple spaces. We remark that the Jini [1] infrastructure o ers similar mechanisms (JavaSpace [21]) for matching client requests and services.

*The Klaim programming paradigm identi es 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 pro- cesses may perform three di erent kinds of basic operations: accessing (possi- bly remote) tuple spaces, sending (possibly remotely) processes for execution and creating new nodes. In particular, mobility can be performed in two dif- ferent 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 2-types, similar to those in 2 [6], are used to classify global values that can freely move over the network.

*We present a simpli ed version of HotKlaim, where pattern matching* is replaced by lambda-abstraction and there is only one operation to read values from tuple spaces, which suÆces 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 de nes 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 2 N is identi ed with the set fi 2 Nji < mg of its predecessors;*

*e denotes a sequence of terms and jej indicates the number of its elements;*

*if 1 and 2 are sequences of declarations, then 1; 2 denotes their con-* catenation (and similarly for sequences of terms);

*(A) is the set of multisets with elements in A, and ] is multiset union;*

*e2[x:= e1] denotes substitution of e1 for x in e2 (modulo alpha-conversion).*

*HotKlaim is a kernel functional language based on system F [10], ex-* tended with the modal types of 2 [6] and Klaim's primitives for concur- rency, 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;:: :;*

*a numerable set X of term variables, ranged over by x; y;:: :;*

*a numerable set L of localities, ranged over by l; l1;:: :;*

*a nite set O = fnil; spawn; new; input; outputg of local operations, ranged over by o.*

*Local operations take a xed number of type- and term-operands*

*Types t 2 T::= X j L j P j t1 ! t2 j (tiji 2 m) j 2t j 8X: k:t with k = g; l Fully global types g 2 G::= L j (giji 2 m) j 2t*

*Terms e 2 E::= x j l j x: t: k:e j e1 e2 j x x: t: k:e j (eiji 2 m) j i(e) j X: k:e j e[t] j nil j o[t](e)*

*where jtj = #t o and jej = #o > 0*

*Signatures fin L*

*Contexts ; 2 Ctx::= ; j ; X: k j ; x: t: k*

*Fig. 1. Syntax for types and terms.*

*process: nil, spawn(e)*

*new locality: new(e)*

*communication: input[t](e1; e2), output[t](e1; e2)*

*#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, nally, nil stands for the terminated process.

*The syntax of HotKlaim is given in Figure 1. The de nition of the set* FV( ) of free variables is as expected, e.g. FV( x: t: k:e) = FV(t) [ (FV(e) fxg).

*In HotKlaim there are two kinds of types (and terms), global- and local-* types (terms), classi ed 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*

*; ` , i.e. is a well-formed context*

*; ` t, i.e. t is a well-formed type*

*; ` e: t, i.e. e is a well-formed term of type t.*

*We will use ; ` J to indicate any of the judgements above. Moreover, we* will use g to indicate the context where local declarations, i.e. those of the form X: l and x: t: l, have been removed. Finally, we will use also the derived judgement ; g `g J to stand for ; g ` 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 ; ` 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: tnil[P ] = P*

*spawn: tspawn[P ] = (() ! P ) ! ()*

*new: tnew[P ] = (L ! P ) ! L*

*input: tinput[P ] = 8X: g:(L; 2X ! P ) ! P*

*output: toutput[P ] = 8X: g:(L; 2X) ! ()*

*For instance, the polymorphic type 8X: g:(L; 2X ! P ) ! 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 func- tion (of type 2X ! 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 x: t: k:e and x x: t: k:e can be declared either local or global, depending on whether k is l or g. When k = l the typing rules are the standard ones.

*The formation rule for 2t requires t to be a global-type. The introduction* and elimination rules for 2t say that 2t is a subset of t (i.e. it classi es a subset of the terms classi ed by t), and that 2t is equal to t (i.e. 2t and t classify the same set of terms) when t 2 G. A key property of 2t is that its values are exactly the global values of type t.

*Only values of type 2t can be exchanged with the input and output op-* erations. In particular, one cannot exchange processes. However, one can

*exchange global values of type Code 8X: l:(to[X]jo 2 O) ! X. Intuitively,*

*Code is the type of process code abstracted over the local type P and the* local operations in O. Its de nition 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 (ojo 2 O) of local operations.

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

*; g `*

*1*

*g*

*t ; 1; X: g; 2 ` J*

*; g `*

*e: t ; 1; x: t: g; 2 ` J*

*; 1; 2[X:= t] ` J[X:= t]*

*1*

*g*

*; 1 ` t ; 1; X: l; 2 ` J*

*; 1; 2[X:= t] ` J[X:= t]*

*; 1; 2 ` J[x:= e]*

*; 1 ` e: t ; 1; x: t: l; 2 ` J*

*; 1; 2 ` J[x:= e]*

*; `*

*X fresh*

*; ` t*

*x fresh*

*; g ` t*

*x fresh*

*; ; X: k `*

*g*

*; ; x: t: l `*

*; ; x: t: g `*

*; ` t1 ; ` t2*

*; ` t1 ! t2*

*; ` ti i 2 m*

*; ` (tiji 2 m)*

*; g ` t*

*; ` 2t*

*g*

*; ; X: k ` t*

*; ` 8X: k:t*

*; `*

*; ` L*

*; `*

*; ` P*

*; `*

*; ` l: L*

*l 2*

*; `*

*; ` x: t*

*x: t: k 2*

*; `*

*; ` e: () ! P*

*; ` e: L ! P*

*; ` nil: P*

*; ` spawn(e): ()*

*; ` new(e): L*

*; g ` t ; ` e : L; e : 2t ! P*

*g 1 2*

*; ` input[t](e1; e2): P*

*; ; x: t1: l ` e: t2*

*; ` ( x: t1: l:e): t1 ! t2*

*; g ` t ; ` e : L; e : 2t*

*; ` output[t](e1; e2): ()*

*; ; x: t1: g ` e: t2*

*g 1 2*

*; ` ( x: t1: g:e): 2t1 ! t2*

*g*

*; ` e1: t1 ! t2 ^ e2: t1*

*; ` e1 e2: t2*

*; ; x: t: l ` e: t*

*; ` x x: t: l:e: t*

*; g ; x: t: g ` e: t*

*; ` x x: t: g:e: 2t*

*; ` ei: ti i 2 m*

*; ` (eiji 2 m): (tiji 2 m)*

*; ` e: (tiji 2 m)*

*; ` i(e): ti*

*i 2 m*

*; g ` e: t*

*g*

*; ` e: 2t*

*g 1*

*; ` e: g*

*; ` e: 2g*

*; ` e: 2t*

*; ` e: t*

*; ; X: k ` e: t*

*; ` ( X: k:e): 8X: k:t*

*; ` e: 8X: l:t2 ; ` t1*

*; ` e[t1]: t2[X:= t1]*

*; ` e: 8X: g:t2 ; g ` t*

*; ` e[t1]: t2[X:= t1]*

*Fig. 2. Type System.*

# *4 Operational semantics*

*An HotKlaim net N 2 Net = (L 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 =) Net (Net + ferrg), written N =) N 0 or N =) err, de ned in terms of a transition relation

*~~>~~ (Sig E A E) + (Sig E ferrg), written ; e a > e0 or*

*Values v 2 V::= l j x: t: k:e j (viji 2 m) j X: k:e*

*Redexes r 2 R::= x j v1 v2 j x x: t: k:e j i(v) j v[t]*

*j o[t](v) where jtj = #t o and jvj = #o > 0*

*Eval. contexts E 2 EC::= [ ] j E e j v E j (v; E; e) j i(E) j E [t]*

*j o[t](v; E; e) where jtj = #t o and jv; ej +1 = #o*

*Fig. 3. Values, redexes and evaluation contexts.*

*; e  ~~>~~ err, describing the potential interactions of a term with the rest of* the net. The set A of potential interactions is

*a 2 A::= j i(v)@l j o(v)@l j s(e) j 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 de ne  ~~>~~ in terms of redexes and evaluation contexts*

*(see Table 3), and a reduction  ~~>~~ (Sig R A E)+ (Sig R ferrg)* 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 input[t](l; v2) corresponds to an* early semantics (according to the terminology used for the -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 de ne the transition relation  ~~>~~ as the least relation* satisfying the following rules:

*a*

*; r > e*

*a*

*; E[r] > E[e]*

*; r  ~~>~~ err*

*; E[r]  ~~>~~ err*

*Finally, the net transition relation =) is given in Figure 5. As in Linda-like* languages, input removes one matching value chosen non-deterministically (since ] 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 2 V cannot be of the form E[r].* For any e 2 E there is at most one E 2 EC and one r 2 R s.t. e E[r], moreover FV(r) FV(e).

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

*Lemma 4.2 (Replacement for Evaluation Contexts)*

*x  ~~>~~ err*

*( x: t: k:e) v2 > e[x:= v2] otherwise v1 v2  ~~>~~ err*

*x x: t: k:e > e[x:= x x: t: k:e]*

*i(viji 2 m) > vi when i 2 m otherwise i(v)  ~~>~~ err*

*( X: k:e)[t] > e[X:= t] otherwise v[t] ~~>~~ err*

*s(v())*

*spawn(v)  ~~>~~ ()*

*l:(v l)*

*; new(v)  ~~>~~ l provided l 62*

*i(v)@l*

*; input[t](l; v2)  ~~>~~ v2 v when l 2 and ; ; ` v: t*

*g*

*; input[t](v1; v2)  ~~>~~ err when v1 62*

*o(v)@l*

*; output[t](l; v)  ~~>~~ () when l 2*

*; output[t](v1; v)  ~~>~~ err when v1 62*

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

*(N ) [ flg; e > e0*

*(N ) [ flg; e  ~~>~~ err*

*N ] (l: e) =) N ] (l: e0 )*

*s(e1)*

*N ] (l: e) =) err*

*l2:e2*

*(N ) [ flg; e  ~~>~~ e2*

*N ] (l: e) =) N ] (l: e1) ] (l: e2)*

*(N ) [ fl1g; e  ~~>~~ e1*

*N ] (l1: e) =) N ] (l1: e1) ] (l2: e2)*

*i(v)@l2*

*(N ) [ fl ;l g; e 0*

*1 2  ~~>~~ e*

*N ] (l : e) ] (l : v) =) N ] (l : e0 ) ] (l : nil)*

*o(v)@l2*

*1 2 1 2*

*(N ) [ fl g; e 0*

*1  ~~>~~ e*

*1 1 2*

*N ] (l : e) =) N ] (l : e0 ) ] (l : v)*

*Fig. 5. Net transition relation. (N ) is the signature flj9e:(l: e) 2 N g.*

*If ; ` E[e]: t with E 2 EC, then there exists t1 such that*

*; ` e: t1*

*0 ; ` e0 : t1 implies 0 ; ` E[e0 ]: t for any 0 and e0*

# *5 Type Safety*

*To state the type safety property we de ne when a net is well-formed.*

*De nition 5.1 A net N is well-formed if, and only if, for every (l: e) 2 N* either (N ); ; ` e: P , or e 2 V and (N ); ; ` e: g for some g 2 G

*(i.e. every e in N is either a local process or a global value)* where (N ) is the signature flj9e:(l: e) 2 N g.

*The following Lemma captures a key property of 2-types.*

*Lemma 5.2 If ; ` v: g, then ; g ` v: g.*

*g*

*Proof. By induction on the derivation of ; ` v: g 2*

*The following Lemmas express type safety for  ~~>~~ and  ~~>~~ .*

*Lemma 5.3 (Safety for  ~~>~~ ) If ; ; ` r: t, then*

*; r  ~~6 >~~ err*

*0 0*

*; r  ~~>~~ e implies ; ; ` e : t*

*s(e2)*

*; r  ~~>~~ e1 implies ; ; ` e2: P and ; ; ` e1: t*

*l:e2*

*; r  ~~>~~ e1 implies l 62 , ; l; ; ` e2: P and ; l; ; ` e1: t*

*i(v)@l 0 0*

*; r  ~~>~~ e implies l 2 , ; ; ` v: g for some g 2 G and ; ; ` e : t*

*o(v)@l 0 0*

*; r  ~~>~~ e implies l 2 , ; ; ` v: g for some g 2 G and ; ; ` e : t*

*Proof. By case analysis on the possible reductions 2*

*Lemma 5.4 (Safety for  ~~>~~ ) If ; ; ` e: t, then*

*; e  ~~6 >~~ err*

*0 0*

*; e  ~~>~~ e implies ; ; ` e : t*

*s(e2)*

*; e  ~~>~~ e1 implies ; ; ` e2: P and ; ; ` e1: t*

*l:e2*

*; e  ~~>~~ e1 implies l 62 , ; l; ; ` e2: P and ; l; ; ` e1: t*

*i(v)@l 0 0*

*; e  ~~>~~ e implies l 2 , ; ; ` v: g for some g 2 G and ; ; ` e : t*

*o(v)@l 0 0*

*; e  ~~>~~ e implies l 2 , ; ; ` v: g for some g 2 G and ; ; ` e : t*

*Proof. If e is active, then it must be of the form E[r]. By Lemma 4.2 there* exists a t1 s.t. ; ; ` r: t1, then one can exploit Safety for  ~~>~~ 2

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

*N =6 ) err*

*N =) N 0 implies that N 0 is well-formed and (N ) (N 0 ).*

*Proof. If N is active, then there is a pair (l: e) 2 N s.t. e is active. Since N* is well-formed, then (N ); ; ` e: P , and one can exploit Safety for  ~~>~~

*2*

# *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::= x!t: k j e j (piji 2 m)

*and ML-like notation for functions and local declarations. Moreover, for* type-setting reasons, we write f g instead of 2 .

*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 ac- cording 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 rst 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 parameter-* ized 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 eld 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 , is considered that results from* the integration of the call-by-value -calculus and the -calculus, together with primitives for process distribution and remote process creation. Di er- ently from HotKlaim, communication is channel based and processes can- not explicitly refer localities (indeed, these are anonymous). D permits the transmission of process abstractions parameterized with respect to resource (i.e. channel) names.

*More speci cally, in [23], a type system for D is de ned 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 di erent avor from that of HotKlaim, sendable values and types play a role similar to our global values and types.

*In [24], a ne-grain type system for D is de ned that permits controlling* the e ect of transmitted process abstractions on local resources (i.e. channels). Di erently from HotKlaim, processes are assigned ne-grain types that, like interfaces, record the resources to which processes have access together with the corresponding capabilities, and process abstractions are assigned depen- dent 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 Con ned- , is presented* that supports distributed computing by allowing expressions at di erent local- ities to communicate via channels. In Con ned- , 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 de ned that guarantees that each value can roam only within the corresponding region. Di erently from HotKlaim, communication is channel based, the transmissible process ab- stractions 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. Schei er, J. Waldo, A. Wollrath, B. O'Sullivan. The Jini speci cation. 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 Speci cations 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. Interpr etation fonctionelle et elimination des coupures de l'arithm etique d'ordre sup erieur. PhD. Thesis, Universit e 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 ow 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. Con ned 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 e ects: 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 e ects: 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 Speci cations. Available at* [*http://java.sun.com/products/javaspaces,*](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/n~yoshida,*](http://www.mcs.le.ac.uk/n~yoshida)