MetaKlaim: Meta-Programming for Global Computing*
(Position Paper)

Gianluigi Ferrari[1], Eugenio Moggi[2], and Rosario Pugliese[3]

[1] Dip. di Informatica, Univ. di Pisa, Italy
[2] Dip. di Informatica e Scienze dell'Informazione, Univ. di Genova, Italy
[3] Dip. di Sistemi e Informatica, Univ. di Firenze, Italy

Abstract. Most foundational models for global computing have focused on the *spatial* dimension of computations, however global computing requires also new ways of thinking about the *temporal* dimension. In particular, with no central control and the need to operate with incomplete information there is a compelling need to interleave meta-programming activities (like assembly and linking of code fragments), security checks (like type-checking at administrative boundaries) and normal computational activities. MetaKlaim is a case study in modeling both spatial and temporal aspects of computing by integrating MetaML (an extension of SML for multi-stage programming) and Klaim (a Kernel Language for Agents Interaction and Mobility). The staging annotations of MetaML provide a fine-grain control of the temporal aspects, while Klaim allows to model and program the spatial aspects of distributed concurrent applications. Our approach for combining these aspects is quite general and should be applicable to other languages/systems for network programming.

1 Introduction

The distributed software architecture (model) which underpins most of the wide area network (WAN) applications typically consists of a large number of heterogeneous computational entities (sometimes referred to as nodes or sites of the network) where components of applications are executed. The various nodes are handled by different authorities having different administrative policies and security requirements. Components of WAN applications are characterized by an highly dynamic behaviour. For instance, a component which acts as a server may become later a client asking for services to other components. Moreover, components have to deal with the unpredictable changes over time of the network environment (changes due to the availability of network connectivity, lack of services, node failures, network reconfiguration, and so on). Finally, nomadic or mobile components may detach from a node and re-attach later on a different

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

node. Hence, components must be designed to support heterogeneity and interoperability. Differently from traditional middle-wares for distributed programming, the structure of the underlying network is made manifest to programmers of WAN applications. We refer to Fuggetta, Picco and Vigna [16] and to Cardelli [5] for a comprehensive analysis of this issue.

The problems associated with the development of WAN applications has prompted the study of the foundations of programming languages with advanced features including mechanisms for agent mobility, for managing security, and for coordinating and monitoring the use of resources. Several foundational calculi have been proposed to tackle most of the phenomena related to WAN programming. We mention the Distributed Join-calculus [17], Klaim [9], the Distributed π-calculus [21], the Ambient calculus [3], the Seal calculus [34], and Nomadic Pict [30]. All these foundational models encompass a notion of location to reflect the idea of administrative domains, and computations at a certain location are under the control of a specific authority. In other words, they focus on the *spatial* dimension of WAN programming[1].

Another crucial aspect of WAN programming concerns the *temporal* dimension: the run-time system may interleave computational activities with structuring activities (e.g. the dynamic assembling of components). Components of WAN applications are often developed and maintained by different providers and may be downloaded on demand. Dynamic linking and dynamic enforcement of security checks (e.g. authentication and access control) increase the flexibility of WAN applications since it makes possible to reconfigure the application without having to restart the application. Several papers have addressed the problem of formally understanding dynamic linking (and separate compilation) [2, 13, 23, 29]; other approaches have tackled the problems of security in systems of mobile agents (see e.g. [12, 20, 22] and the references therein).

Hence, the spatial and the temporal dimension of WAN programming have been studied at considerable depth but in *isolation* and their interplay has not yet properly formalized and understood. This paper attempts to develop a foundational model which integrates together both the spatial and the temporal aspects of WAN programming. We have abstracted the basic feature of the problem in a calculus having primitives for programming agents which may migrate among sites, and primitives which support fine-grain control of dynamic linking and security checks.

Our calculus builds on the Klaim language [9] and the MetaML functional language. The language Klaim (Kernel Language for Agents Interaction and Mobility) is an experimental programming language, inspired by the Linda coordination model [19, 6], specifically designed to model and to program WAN applications by exploiting mobility. Klaim provides direct support for expressing and enforcing access control policies to resources and for authorizing migration and execution of mobile processes [12, 11]. MetaML [25] supports most features of SML and meta-programming constructs. Meta-programming provides

[1] The spatial dimension of WAN programming is often referred to as *network awareness*.

an ideal tool for describing customization and combination of software components. In other words, MetaML enables programming of a variety of structural re-arrangements of code *directly* since the meta-programming constructs have the same status of the other programming constructs.

Sections 2 and 3 give an high-level overview of MetaML and HotKlaim (a higher-order variant of Klaim) introduced in Ferrari, Moggi and Pugliese [15]. Section 4 describes the formal development of MetaKlaim, a core language that integrates the meta-programming features of MetaML with the programming constructs of Klaim, and discusses how MetaKlaim may handle some of the most relevant issues raised by global computing. Section 5 gives few programming examples particularly relevant in a global computing scenario.

2 MetaML

MetaML [25] is a substantial language, supporting most features of SML and a host of meta-programming constructs. In the current public release, safety is guaranteed only for programs in the pure fragment. Most of the theoretical work has gone into establishing type safety for larger subsets of the language. The key idea in multi-stage programming is the use of *annotations* to allow the programmer to breakdown the cost of a computation into distinct *stages*. MetaML provides a type constructor $\langle\_\rangle$ for code fragments with potentially unresolved links (represented by *dynamic variables*), and three basic staging annotations: Brackets $\langle\_\rangle$, Escape ~ _ and Run run _. Brackets defers the computation of its argument (constructing code instead); Escape splices its code argument into the body of surrounding Brackets (combining code fragments into a larger fragment); and Run executes its code argument. The following examples illustrate more concretely how the staging annotation affects evaluation:

```
-| val a = <1>;
val a = <1> : <int>
-| val b = <~a+~a>;
val b = <1+1> : <int>
-| val c = run b;
val c = 2 : int
```

A distinguished feature of multi-stage (and multi-level) languages is the ability to "evaluate under a dynamic binder", and to manipulate values with free "dynamic variables" at run-time. For instance:

```
-| fun double x = <~x+~x>;
val double = fn ... :  <int> -> <int>
-| val f = <fn x => ~(double <x+1>)>;
val f = <fn x => (x+1)+(x+1)> : <int -> int>
```

the value `<x+1>` has a dynamic variable `x` free, and evaluation of `double <x+1>` is performed within the scope of a dynamic binder `<fn x =>...>`. Having values with free dynamic variables complicates the typing of *run* and of the three

operations on references. In fact, the naive typing of such operations does not ensure type safety, as the following examples show:

```
-| <fn x => ~(run <x>;<x+1>)>;
(* x is an unresolved link, thus one cannot execute x *)
-| val l = ref <1>;
val l = ... : <int> ref;
-| val f = <fn x => ~(l:=<x>;<x+1>)>;
val f = <fn x => x+1> : <int -> int>
(* l contains <x>, so x is outside the scope of its binder *)
```

One of our starting points for designing MetaKlaim is the work by Calcagno, Moggi, Sheard and Taha [1], which exploits *closed types* to ensure type safety of MetaML's staging annotations and cross-stage persistence in the presence of SML-style references. In fact, the authors conjecture that closed types provide a general solution for safely adding multi-stage programming constructs in the presence of computational effects. The key property of a term e of closed type is that "all free occurrences in e of dynamic variables are *dead code*", and therefore they can be ignored. The closed type constructor $[\_]$, first introduced in Moggi, Taha, Benaissa and Sheard [26] for typing Run, maps a type t to the *biggest* closed type $[t]$ included in t.

Note 1. Hereafter, we use the following notations and conventions:

- Term equivalence, written $\equiv$, is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e. If E is a set of terms, then E_0 indicates the set of terms in E without free variables. Substitution of e_1 for x in e_2 (modulo $\equiv$) is written $e_2[x := e_1]$.
- m, n range over the set N of natural numbers, and $m \in \mathsf{N}$ is identified with the set of its predecessors.
- $\overline{e}$ denotes a sequence of terms and $|\overline{e}|$ indicates the number of its elements.
- $f : A \stackrel{fin}{\to} B$ means that f is a partial function from A to B with a finite domain, written $dom(f)$.
- $\mu(A)$ is the set of multisets with elements in A, and $\uplus$ denotes multiset union.
- Given an declaration of a grammar such as $e := P_1 \mid \ldots \mid P_m$, we write $e+ = P_{m+1} \mid \ldots \mid P_{m+n}$ as a shorthand for $e := P_1 \mid \ldots \mid P_{m+n}$.

Figure 1 summarizes the syntax of terms, types and closed types for the fragment of MetaML considered in Calcagno, Moggi, Sheard and Taha [1]. Variables x range over an infinite set X (and locations l over an infinite set L).

The type system of MetaML uses judgments of the form $\Delta; \Gamma \vdash_n e : t$, read "$e$ has type t at level n", where $\Delta : \mathsf{X} \stackrel{fin}{\to} (\mathsf{C} \times \mathsf{N})$ and $\Gamma : \mathsf{X} \stackrel{fin}{\to} (\mathsf{T} \times \mathsf{N})$ are type-and-level assignments. The level information is typical of type systems for multi-level languages (like $\lambda^{\bigcirc}$ of Davies [7]), in this context dynamic variables correspond to variables declared at level > 0. The splitting of type-and-level assignments in two parts (Δ and Γ) is borrowed from $\lambda^{\Box}$ [8], and is needed for typing terms of type $[t]$. More formally, the difference between a declaration in

- Terms $e \in \mathsf{E} ::= x \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}\ x.e$ functional fragment
  $\mid\ l \mid ref(e) \mid get(e) \mid set(e_1, e_2)$ references
  $\mid\ \langle e \rangle \mid \tilde{}e \mid \%e \mid run(e)$ meta-programming
  $\mid\ (x)e$ dead-code annotation
  Bind $(x)e$ declares that the free occurrences of x in e are *dead code*. Bind is used in the operational semantics for handling *scope extrusion*.
- Types $t \in \mathsf{T} ::= \ldots \mid t_1 \to t_2 \mid [t] \mid \mathsf{ref}\ c \mid \langle t \rangle$
  Closed types $c \in \mathsf{C} ::= \ldots \mid t_1 \to c_2 \mid [t] \mid \mathsf{ref}\ c$
  where $\ldots$ stands for some unspecified base types, e.g. `int`.

Fig. 1. Syntax of METAML types and terms

Δ and the same declaration in Γ is expressed by the following substitution rules:

$$\frac{\Sigma; \Delta; \Gamma \vdash_m e_1 : t_1 \quad \Sigma; \Delta; \Gamma, x : t_1^m \vdash_n e_2 : t_2}{\Sigma; \Delta; \Gamma \vdash_n e_2[x := e_1] : t_2} \qquad \frac{\Sigma; \Delta^{\leq m}; \emptyset \vdash_m e_1 : c_1 \quad \Sigma; \Delta, x : c_1^m; \Gamma \vdash_n e_2 : t_2}{\Sigma; \Delta; \Gamma \vdash_n e_2[x := e_1] : t_2}$$

Figure 2 recalls the typing rules for the pure fragment, in particular:

- $\%e$ allows to use at higher levels a value defined at level n, this feature is called *cross-stage persistence*;
- $run(e)$ allows to execute the *complete* program (i.e. without unresolved links) represented by e. Since $run(e)$ is at the same the level of e, one could considered run as a constant of type $[\langle t \rangle] \to [t]$.

Given that one allows only references to values of a closed type, the three SML operations on references have the expected types, namely:

$$ref : c \to \mathsf{ref}\ c \qquad get : \mathsf{ref}\ c \to c \qquad set : \mathsf{ref}\ c \to c \to \mathsf{ref}\ c$$

Calcagno, Moggi, Sheard and Taha [1] adopt a small-step operational semantics, which eases the transfer of definitions and results to a concurrent setting. More precisely, there are transition relations $e \overset{n}{\longmapsto} e'$, one for each level, defined in terms of evaluation contexts [35] and two reductions $\overset{i}{\longrightarrow}$, one at level 0 (normal evaluation) and the other at level 1 (symbolic evaluation):

$$\frac{r \overset{i}{\longrightarrow} e'}{E_i^n[r] \overset{n}{\longmapsto} E_i^n[e']} \quad E_i^n \text{ evaluation context at level } n \text{ with hole at level } i$$

In fact, one is only interested in transitions $e \overset{0}{\longmapsto} e'$, where e is closed. However, the ability to evaluate under dynamic binders means that a redex r could have free dynamic variables even when $E_i^n[r]$ is closed. The following are the

$$\frac{}{\Delta;\Gamma \vdash_n x : t}\ (\Delta,\Gamma)(x) = t^n \qquad \frac{\Delta;\Gamma, x : t_1^n \vdash_n e : t_2}{\Delta;\Gamma \vdash_n \lambda x.e : t_1 \to t_2}$$

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash_n e_1 : t_1 \to t_2\\ \Delta;\Gamma \vdash_n e_2 : t_1\end{array}}{\Delta;\Gamma \vdash_n e_1 e_2 : t_2} \qquad \frac{\begin{array}{l}\Delta;\Gamma \vdash_n e_1 : c_1\\ \Delta, x : c_1^n;\Gamma \vdash_n e_2 : t_2\end{array}}{\Delta;\Gamma \vdash_n \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2} \qquad \frac{\Delta;\Gamma, x : t^n \vdash_n e : t}{\Delta;\Gamma \vdash_n \mathsf{fix}\ x.e : t}$$

$$\frac{\Delta;\Gamma \vdash_{n+1} e : t}{\Delta;\Gamma \vdash_n \langle e\rangle : \langle t\rangle} \qquad \frac{\Delta;\Gamma \vdash_n e : \langle t\rangle}{\Delta;\Gamma \vdash_{n+1} \tilde{}e : t} \qquad \frac{\Delta;\Gamma \vdash_n e : t}{\Delta;\Gamma \vdash_{n+1} \%e : t} \qquad \frac{\Delta;\Gamma \vdash_n e : [\langle t\rangle]}{\Delta;\Gamma \vdash_n \mathsf{run}\ e : [t]}$$

$$\frac{\Delta;\Gamma \vdash_n e : t}{\Delta;\Gamma \vdash_n (x)e : t}\ x \text{ fresh} \qquad \frac{\Delta;\Gamma, x : t_1^m \vdash_n e : c}{\Delta;\Gamma \vdash_n (x)e : c}\ m > n \qquad \frac{\Delta, x : c_1^m;\Gamma \vdash_n e : c}{\Delta;\Gamma \vdash_n (x)e : c}\ m > n$$

$$\frac{\Delta;\Gamma \vdash_n e : c}{\Delta;\Gamma \vdash_n e : [c]} \qquad \frac{\Delta^{\leq n};\emptyset \vdash_n e : t}{\Delta;\Gamma \vdash_n e : [t]} \qquad \frac{\Delta;\Gamma \vdash_n e : [t]}{\Delta;\Gamma \vdash_n e : t}$$

Fig. 2. Type System for the pure fragment of MetaML.

reductions for the pure fragment

$$((X)\lambda x.e)\ v^0 \xrightarrow{0} ((X)e)[x := v^0] \qquad \mathsf{fix}\ x.e \xrightarrow{0} e[x := \mathsf{fix}\ x.e]$$

$$\mathsf{let}\ x = v^0\ \mathsf{in}\ e \xrightarrow{0} e[x := \bullet v^0] \qquad run((X)\langle v^1\rangle) \xrightarrow{0} (X)v^1 \downarrow_0$$

$$\tilde{}(X)\langle v^1\rangle \xrightarrow{1} (X)v^1$$

where $(X)e$ is iterated Bind, $\bullet e$ is the Bind-closure of e (i.e. all free variables in e get bound by Bind in $\bullet e$), $v^{n+1} \downarrow_n$ is a Demotion operation which turns values at level $n+1$ into terms, and v^n ranges over the set $\mathsf{V}^n \subset \mathsf{E}$ of values at level n

$$v^0 \in \mathsf{V}^0 ::= l \mid \lambda x.e \mid \langle v^1\rangle \mid (x)v^0$$
$$v^{n+1} \in \mathsf{V}^{n+1} ::= \ldots$$

In the reduction $\mathsf{let}\ x = v^0\ \mathsf{in}\ e \xrightarrow{0} e[x := \bullet v^0]$ we substitute x with $\bullet v^0$ rather than v^0. This can be done because the type of v^0 must be closed, and so all dynamic variables in v^0 are dead code. For the same reason, the reductions for $ref(v^0)$ and $set((X)l, v^0)$, which are the redexes that cause an update of the store, write to the store $\bullet v^0$ rather than v^0. In this way one ensures that the store contains only closed values, and that any free dynamic variable in v^0, that would go outside the scope of its dynamic binder, is re-bound by Bind.

3 Klaim

We now briefly reviews the main features of Klaim by presenting a variant, called HotKlaim (for Higher-order typed Klaim), which is more suitable for integra-

- Types $t \in \mathsf{T} ::= L \mid t_1 \to t_2 \mid (t_i | i \in m)$ $(m \geq 0)$
- Terms $e \in \mathsf{E} ::= x \mid (mr_i | i \in n) \mid e_1 e_2 \mid \mathsf{fix}\, x.e$ functional fragment $(n > 0)$
  $\mid (e_i | i \in m) \mid l \mid op(\overline{e})$ $(m \geq 0)$ and $(|\overline{e}| = \#op)$
  where op ranges over a set $\mathsf{Op} = \{nil, new, spawn, input, read, output\}$ of primitive operations. The arities of these operations are: $\#nil = 0$, $\#new = \#spawn = 1$ and $\#input = \#read = \#output = 2$.
  Patterns $p \in \mathsf{P} ::= x!t \mid e \mid (p_i | i \in m)$ $(m \geq 0)$
  Match Rules $mr ::= p \Rightarrow e$
- Values $v \in \mathsf{V} ::= l \mid (vmr_i | i \in n) \mid (v_i | i \in m)$ $(n > 0$ and $m \geq 0)$
  Evaluated Patterns $vp \in \mathsf{VP} ::= x!t \mid v \mid (vp_i | i \in m)$ $(m \geq 0)$
  Evaluated Match Rules $vmr ::= vp \Rightarrow e$

Fig. 3. Syntax of HotKlaim types, terms and values

tion with MetaML. Figure 3 summarizes the syntax of HotKlaim terms, which extends that of a core functional language with pattern matching[2].

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 (TS), i.e. a multi-set of tuples. Processes are distributed over nodes and communicate asynchronously via TSs. The types of HotKlaim include the type L of localities and tuple types $(t_i | i \in m)$.

- nil is the deadlock process, while $spawn(e)$ activates a process in a parallel thread. These operations have type $nil : t$ and $spawn : (() \to t) \to ()$ respectively.
- $output(l, v)$ adds v to the TS at l ($output$ is non-blocking). The type of $output$ is $(L, t) \to ()$.
- $input(l, (vp_i \Rightarrow e_i | i \in m))$ and $read(l, (vp_i \Rightarrow e_i | i \in m))$ access the TS located at l. $input$ checks each value pattern vp_i and looks in the TS at l for a matching v. If such a v exists, it is removed from the TS, and the variables declared in the matching pattern vp_j (i.e. those indicated by $x!t$) are replaced within e_j by the corresponding values in v. If no matching tuple is found, the operation is suspended until one becomes available (thus $input$ is a blocking operation). $read$ differs from $input$ only in that the matching v is not removed from the TS. The type of $input$ and $read$ is $(L, t_1 \to t_2) \to t_2$. These are the only operations using dynamic type-checking (namely a matching v must be consistent with the types attached to variables declared in a pattern).
- $new(e)$ creates a new locality l, activates process $e\ l$ at l, and returns the new locality. Therefore, the type of new is $(L \to t) \to L$.

In Klaim there is also an operation $eval(l, e)$ for process mobility, which activates a process at locality l. The operation $eval$ has not been included in HotKlaim. In fact, $eval$ relies on dynamic scoping (a potentially dangerous

[2] The patterns are more general than those of SML, namely they include terms (that should evaluate to a locality).

mechanism), which is not available in HotKlaim, since in a functional language one can use (the safer mechanism of) parameterization. Moreover, the form of mobility underlying *eval* is "asynchronous", i.e. it involves only the sending node, this can be a source of security problems, because the target node has no control over the incoming processes. HotKlaim allows only "synchronous" process mobility. More precisely, a (sending) node can *output* a process abstraction in any TS, but the abstraction can become an active process only if (a process at) another (receiving) node does *input*/*read* it. Without *eval*, remote communication between nodes, like that provided by Klaim primitives, is essential to implement mobility.

A HotKlaim *net* $N \in \mathsf{Net} \stackrel{\Delta}{=} \mu(\mathsf{L} \times \mathsf{E}_0)$ is a multi-set of pairs 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 TS of that node). The dynamics of a net is given by a relation $N \Longrightarrow N'$ defined in terms of transition relations $e \stackrel{a}{\longmapsto} e'$ for terms, where a ranges over the set of *potential interactions*

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

For instance, $i(v)@l$ is the capability of *inputing* a value v from the TS at l, while $l : e$ is the (non-blocking) action of creating a new locality l running process e. The transitions relation $\stackrel{a}{\longmapsto}$ is defined in terms of evaluation contexts and a corresponding reduction $\stackrel{a}{\longrightarrow}$. The following is a sample of the reductions

$$\begin{array}{ll}
(vp_i \Rightarrow e_i | i \in n)\ v \stackrel{\tau}{\longrightarrow} e_j[\rho] & \text{if } j \in n \text{ and } match(vp_j, v) = \rho \\
 & \text{and } \forall i < j. match(pv_i, v) = fail \\
(vp_i \Rightarrow e_i | i \in n)\ v \stackrel{\tau}{\longrightarrow} fail & \text{if } \forall i \in n. match(pv_i, v) = fail \\
input(l, (vp_i \Rightarrow e_i | i \in n)) \stackrel{i(v)@l}{\longrightarrow} e_j[\rho] & \text{if } j \in n \text{ and } match_t(vp_j, v) = \rho \\
 & \text{and } \forall i < j. match_t(pv_i, v) = fail
\end{array}$$

$$output(l, v) \stackrel{o(v)@l}{\longrightarrow} () \qquad spawn(v) \stackrel{s(v())}{\longrightarrow} () \qquad new(v) \stackrel{l:v\ l}{\longrightarrow} l$$

the functions $match(vp, v)$ and $match_t(vp, v)$ check the value v against the value pattern vp, and either return *the* matching substitution (if it exists) or $fail$, e.g. $match(v, v') = \emptyset$ if $v \equiv v' \in \mathsf{L}$, otherwise $fail$. The function $match_t$ does in addition dynamic type-checking, i.e. $match_t(x!t, v) = [x := v]$ when $\emptyset \vdash v : t$, otherwise $fail$.

4 MetaKlaim: a Proposal

This Section describes MetaKlaim, which integrates the meta-programming features of MetaML with HotKlaim. The integration of types and terms is straightforward (see Figure 4). The typing judgments are those of MetaML, and the type system extends that for the pure fragment of MetaML (see Figure 2). The most critical design choice concerns the operational semantics:

– One choice is to have a sharp separation between symbolic evaluation and process interaction, namely to forbid process interaction during symbolic evaluation (i.e. within the scope of a dynamic binder).
– The other choice is to allow arbitrary interleaving of symbolic evaluation and process interaction.

We have opted for the second choice, since it offers a higher degree of flexibility, but one must be careful in typing the primitive operations $op \in \mathsf{Op}$, in order to prevent scope extrusion when process interaction occurs within the scope of a dynamic binder. The solution is similar to that for the SML operations on references, in certain cases one must use closed types instead of arbitrary types:

$$\begin{array}{lll} nil : t & new : (L \to c) \to L & spawn : (() \to c) \to () \\ input, read : (L, c \to t) \to t & & output : (L, c) \to () \end{array}$$

The transition relations $e \stackrel{a,n}{\longmapsto} e'$ are defined in terms of evaluation contexts (see Figure 5) and reductions $r^0 \stackrel{a}{\longrightarrow} e'$ (for interactions) and $r^1 \stackrel{1}{\longrightarrow} e'$ (for symbolic evaluation), namely

$$\frac{r^0 \stackrel{a}{\longrightarrow} e'}{E_0^n[r^0] \stackrel{a,n}{\longmapsto} E_0^n[e']} \qquad \frac{r^1 \stackrel{1}{\longrightarrow} e'}{E_1^n[r^1] \stackrel{\tau,n}{\longmapsto} E_1^n[e']}$$

In fact, for defining the dynamics of a net only transition relations $\stackrel{a,n}{\longmapsto}$ with $n = 0$ are needed.

The reductions for the primitive operations $op(\overline{e})$ have to be modified as follows in order to prevent scope extrusion:

$$\begin{array}{ll} input(l, (vp_i^0 \Rightarrow e_i | i \in n)) \stackrel{i(v^0)@l}{\longrightarrow} e_j[\rho] & \text{if } j \in n \text{ and } match_t(vp_j^0, v^0) = \rho \\ & \text{and } \forall i < j. match_t(pv_i^0, v^0) = fail \end{array}$$

$$output(l, v^0) \stackrel{o(\bullet v^0)@l}{\longrightarrow} () \qquad spawn(v^0) \stackrel{s(\bullet v^0())}{\longrightarrow} () \qquad new(v^0) \stackrel{l:\bullet v^0\ l}{\longrightarrow} l$$

The definition of net is like in HotKlaim, i.e. a multi-set $N \in \mathsf{Net} \stackrel{\Delta}{=} \mu(\mathsf{L} \times \mathsf{E}_0)$ of pairs. The following is a sample of the net transition rules

$$\frac{e \stackrel{i(v)@l_2,0}{\longmapsto} e'}{N \uplus (l_1 : e) \uplus (l_2 : v) \Longrightarrow N \uplus (l_1 : e') \uplus (l_2 : nil)}$$

$$\frac{e \stackrel{o(v)@l_2,0}{\longmapsto} e'}{N \uplus (l_1 : e) \Longrightarrow N \uplus (l_1 : e') \uplus (l_2 : v)}\ l_2 \in L(N) \cup \{l_1\}$$

$$\frac{e \stackrel{l_2:e_2,0}{\longmapsto} e_1}{N \uplus (l_1 : e) \Longrightarrow N \uplus (l_1 : e_1) \uplus (l_2 : e_2)}\ l_2 \notin L(N) \cup \{l_1\}$$

where $L(N) \stackrel{\Delta}{=} \{l | \exists e.(l : e) \in N\} \subseteq_{fin} \mathsf{L}$ is the set of nodes in the net N. We say that a net N is well-formed $\stackrel{\Delta}{\Longleftrightarrow}$ for every $(l : e) \in N$ exists $c \in \mathsf{C}$ s.t. $\emptyset; \emptyset \vdash_0 e : c$ and all localities occurring in e are in $L(N)$, i.e. are nodes of the net.

- Types $t \in \mathsf{T} ::= L \mid t_1 \to t_2 \mid (t_i | i \in m) \mid [t] \mid \langle t \rangle \ (m \geq 0)$
  Closed types $c \in \mathsf{C} ::= L \mid t_1 \to c_2 \mid (c_i | i \in m) \mid [t] \ (m \geq 0)$
- Terms $e \in \mathsf{E} ::= x \mid (mr_i | i \in n) \mid e_1 e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}\ x.e \quad (n > 0)$
  $\mid (e_i | i \in m) \mid l \mid op(\overline{e}) \qquad (|\overline{e}| = \#op \text{ and } m \geq 0)$
  $\mid \langle e \rangle \mid \tilde{}e \mid \%e \mid run(e) \mid (x)e$
  Patterns $p \in \mathsf{P} ::= x!t \mid e \mid (p_i | i \in m) \ (m \geq 0)$
  Match Rules $mr ::= p \Rightarrow e$
- Values $v^n \in \mathsf{V}^n \subset \mathsf{E}$ at level $n \in \mathsf{N}$
  $v^0 ::= l \mid (vmr_i^0 | i \in n) \mid (v_i^0 | i \in m) \mid \langle v^1 \rangle \mid (x)v^0$
  $v^{n+1} ::= x \mid (vmr_i^{n+1} | i \in n) \mid v_1^{n+1} v_2^{n+1} \mid \mathsf{let}\ x = v_1^{n+1}\ \mathsf{in}\ v_2^{n+1} \mid \mathsf{fix}\ x.v^{n+1}$
  $\mid (v_i^{n+1} | i \in m) \mid l \mid op(\overline{v}^{n+1})$
  $\mid \langle v^{n+2} \rangle \mid \%v^n \mid run(v^{n+1}) \mid (x)v^{n+1}$
  $v^{n+2} + = \tilde{}v^{n+1}$
  Evaluated Patterns $vp^n \in \mathsf{VP}^{\cdot} := x!t \mid v^n \mid (vp_i^n | i \in m)$
  Evaluated Match Rules $vmr^0 ::= vp^0 \Rightarrow e$
  $vmr^{n+1} ::= vp^{n+1} \Rightarrow v^{n+1}$

Fig. 4. Syntax of MetaKlaim types, terms and values

We expect that the two main results established in Calcagno, Moggi, Sheard and Taha [1], namely type safety for MetaML and that MetaML is a conservative extension of SML, extend smoothly to MetaKlaim. In particular, we have:

Theorem 1 (Type Safety). *If N is a well-formed net, then*

- $N \not\Longrightarrow \mathsf{err}$
- $N \Longrightarrow N'$ *implies that* N' *is well-formed and* $L(N) \subseteq L(N')$.

4.1 MetaKlaim and Global Computing

Global computing demands programming applications with different degrees of computational requirements and specific restrictions over resources. At the programming level, what is needed are constructs which permit programming and deploying a variety of computational policies to respond to the evolving demands of the run-time environment. In this paper, we outlined the development of MetaKlaim a basic core language for programming the temporal and spatial dimensions of global computing. The possibility of interaleaving metaprogramming activities with computational activities gives to MetaKlaim programmers the ability of programming policies without requiring a deep knowledge of the underlying system infrastructure. Indeed, the ability of directly accessing code fragments provides a high flexibility. For instance, one can program policies that constraint resource usages without rewriting the code of resource libraries. Moreover, platform independent code transformations can be programmed as well.

– Redexes $r^i \in \mathsf{R}^i$ at level $i \in \{0,1\}$

$$r^0 ::= x \mid v_1^0 v_2^0 \mid \mathsf{fix}\ x.e \mid \mathsf{let}\ x = v^0\ \mathsf{in}\ e$$
$$\mid\ op(\overline{v}^0) \qquad op \not\equiv nil \text{ and } |\overline{v}^0| = \#op$$
$$\mid\ \tilde{}e \mid \mathsf{run}\ v^0 \mid \%e$$

$$r^1 ::= \tilde{}v^0$$

– Evaluation Contexts $E_i^n \in \mathsf{EC}_i^n$ at level $n \in \mathsf{N}$ with hole at level $i \in \{0,1\}$

$$E_i^n ::= (\overline{vmr}^n, Ep_i^n \Rightarrow e, \overline{mr}) \mid E_i^n e \mid v^n E_i^n \mid \mathsf{let}\ x = E_i^n\ \mathsf{in}\ e$$
$$\mid\ (\overline{v}^n, E_i^n, \overline{e}) \mid op(\overline{v}^n, E_i^n, \overline{e}) \qquad |\overline{v}^n| + 1 + |\overline{e}| = \#op$$
$$\mid\ \langle E_i^{n+1} \rangle \mid \mathsf{run}\ E_i^n \mid (x)E_i^n$$

$$E_i^{n+1} += (\overline{vmr}^{n+1}, vp^{n+1} \Rightarrow E_i^{n+1}, \overline{mr}) \mid \mathsf{let}\ x = v^{n+1}\ \mathsf{in}\ E_i^{n+1} \mid \mathsf{fix}\ x.E_i^{n+1}$$
$$\mid\ \tilde{}E_i^n \mid \%E_i^n$$

$$E_i^i += []$$

Evaluation Contexts for patterns $Ep_i^n ::= E_i^n \mid (\overline{vp}^n, Ep_i^n, \overline{p})$

– Interactions $a \in \mathsf{A} ::= \tau \mid l : e \mid s(e) \mid i(v^0)@l \mid r(v^0)@l \mid o(v^0)@l$
with $\mathrm{FV}(e) = \mathrm{FV}(v^0) = \emptyset$

Fig. 5. MetaKlaim's redexes and evaluation contexts

There are several aspects, that are very important for global computing, which are not adequately handled in this preliminary proposal, e.g. genericity of SW components and security for systems of mobile components.

Genericity. In a global computing scenario most SW components available on the network are expected to be highly parameterized. Functional abstraction is not enough to express the desirable forms of parameterization. Also a limited form of polymorphism, like that supported by SML, appears inadequate. For instance, the full power of system F [18] is needed for expressing the type of mobile process abstractions used in the more sophisticated implementations of the *nomadic data collector* as we will briefly discuss in Section 5.

Security. The dynamic type-checking performed by the *input* and *read* primitives of Klaim provide a simple and effective mechanism to ensure that a value fetched from a TS meets certain requirements. However, the expressiveness of this mechanism is directly related to the expressiveness of the type system. For instance, one would like to check that a process abstraction fetched from a TS does not perform unwanted interactions (e.g. communication with a certain locality) or meta-programming activities (e.g. symbolic evaluation or code execution). We believe that a promising approach is to adopt a type-and-effect system [32, 33, 27].

5 Examples

We have already pointed out components available on the network are expected to be highly parameterized, in order to accommodate a multiplicity of applica-

tions and to adapt to a variety of platforms and environments. A way to reconcile genericity and efficiency is to use *generative* components, which embody a method for constructing efficient object-code, once most of the parameters for the component have been fixed. Kamin, Callahan and Clausen [24] give numerous examples of components for generating object-code (for instance Java). These components are described as higher-order macros in a *functional* meta-language with Bracket and Escape constructs similar to those of MetaML. We give two specific examples, a linker and a nomadic data collector, that make use of the process operations of HotKlaim and exemplify the additional advantages of MetaKlaim over HotKlaim.

5.1 Dynamic Linking and Loading

In global computing programming one important issue is the ability to control the loading policy of SW components. For instance, the JVM supports dynamic linking and loading of classes [28, 14]. In some cases (localities with good connectivity or thrusted localities) one wants to load components just-when-needed, in other cases one may prefer to fetch in advance all components requested by a certain application. The naive solution is to parameterized applications w.r.t. a linker, and call the linker whenever a component (or service) is needed. This does not ensure enough flexibility, a better approach is to define a generative component parameterized w.r.t. a meta-linker. The meta-linker can decide whether to load a requested component at code-generation-time or to postpone the loading at run-time, namely by generating code for a call to the naive linker.

In the following example we assume that an application is parameterized w.r.t. a linker, which given a name of a service either succeeds in establishing a connection between the service and the application by returning an authorization key, or raises an exception. We are not interested in the details of the linker, but an abstract behavior could be: check whether the service (or its proxy) is present locally, if not search for it remotely and copy it locally (or create a proxy). We make use of the following types:

```
L       localities
Proc    processes (* e.g. the datatype with no values *)
Name    service names
Key     authorization keys
(* parameterized applications *)
Linker = Name -> Key       (* linkers *)
App = Linker -> Proc
(* parameterized application code *)
MLinker = Name -> <Key>    (* meta-linkers *)
CApp = MLinker -> <Proc>
```

The following process fetches applications from the local tuple place and execute them by passing a linker

```
execute (self:L, linker:Linker): Proc =
fix exec:Proc. input(self, fn x!App => spawn(x linker, exec))
```

If during execution the application calls the linker `link n` and the linker fails to make a connection to the service named `n`, then an exception is raised (and the application stops). The following process works similarly, but fetches application code and use a meta-linker:

```
Mexecute (self:L, mlinker:MLinker): Proc =
fix exec: Proc.
    input(self, fn x![CApp] => spawn(run(x Mlinker), exec))
```

An invocation of the meta-linker will be of the form `<...~(mlinker n)...>`. Using the meta programming facilities, the meta-linker can decide whether to invoke the linker immediately, i.e. `mlinker n = <%(linker n)>`, or whether to generate code for invoking the linker, i.e. `mlinker n = <%linker %n>`. In the first case, when the linker fails to make a connection, the code will not be executed at all.

5.2 Nomadic Data Collector

In this section, we exemplify the use of *mobile code* by means of a simple distributed information retrieval application. We assume that each node of a distributed database contain tuples of the form `(i,d)`, where `i` is the search key and `d` is the associated data, or of the form `(i,l)`, where `l` is a locality where more data associated to `i` can be searched. We give three implementations: two in HotKlaim, and the third in MetaKlaim. Hereafter we make use of the previously introduced types and of the following additional types:

```
Data    search keys and data
(* simple process abstractions *)
PA = L -> Proc
(* polymorphic types of process operations *)
Read = Input = V X:C. V Y:T. (L,X->Y) -> Y
Output = V X:C. V Y:T. (L,X,Y) -> Y
Spawn = V X:C. V Y:T. (X,Y) -> Y
Nil = V Y:T. Y
(* process abstractions with security checks *)
EnvK = (L,Key->Read,Key->Input,Key->Output,Nil,Spawn)
PAK = EnvK -> Proc
(* polymorphic types of process meta-operations *)
MSpawn = V X:C. V Y:T. (<X>,<Y>) -> <Y>
            (* and similarly for the other types *)
(* code abstractions with static security checks *)
MEnvK = (<L>,Key->MRead,Key->MInput,Key->MOutput,MNil,MSpawn)
CAK = MEnvK -> <Proc>
```

Simple process abstraction in HotKlaim. `ppa(i,u)` is a process abstraction activated by a process `execute`. `execute` fetches process abstractions from the local tuple space and activate them by providing the locality of the node. The

parameter `i` is a search key, while `u` is the locality where all data associated to `i` should be sent. After activation `ppa(i,u)` removes data locally associated to `i` and forwards it to `u`, moreover `ppa(i,u)` sends copies of itself to localities that may contain data associated to `i`:

```
ppa(i:Data, u:L) : PA =
fix pa:PA. fn self!L =>
  spawn(fix p:Proc. input(self, fn (i, x!Data) => output(u, x, p)),
        fix p:Proc. input(self, fn (i, l!L) => output(l, pa, p)))

execute (self:L) : Proc =
fix exec:Proc. input(self, fn X!PA => spawn(X env, exec))
```

Process abstraction with security checks in HotKlaim. This implementation uses more complex process abstractions, namely a process is abstracted also w.r.t. surrogate process operations. Moreover, the surrogate communication primitives require an extra parameter (an authorization key):

```
ppa(k:Key, i:Data, u:L) : PAK =
fix pa:PAK. fn (self', _, in' , out', _, spawn'):EnvK =>
  spawn'(fix p:Proc.
             in' k (self', fn (i, x!Data) => out' k (u, x, p)),
         fix p:Proc.
             in' k (self', fn (i, l!L) =>  out' k (l, pa, p)))

execute (self:L, env:EnvK) : Proc =
fix exec:Proc. input(self, fn (X!PAK) => spawn(X env, exec))
```

Code abstraction with static security checks in MetaKlaim. This implementation refines the previous one by exploiting the meta-programming constructs of MetaKlaim, and has advantages similar to those offered by a meta-linker. For instance, depending on the key `k`, the meta-operation `in' k` could generate an `input` with no run-time overhead, or a deadlock `nil` (when the key does not allow to read anything), or some customized run-time checks on what is read.

```
pca(k:Key, i:Data, u:L) : CAK =
fix ca:CAK. fn (self', _, in' , out', _, spawn'):MEnvK =>
  spawn'(<fix p:Proc. ~(in' k (self', <fn (%i, x!Data) =>
                                       ~(out' k (<%u>, <x>, <p>))>))> ,
         <fix p:Proc. ~(in' k (self', <fn (%i, l!L) =>
                                       ~(out' k (<l>, <%ca>, <p>))>))>)

execute (self:L, env:[MEnvK]) : Proc =
fix exec:Proc. input(self, fn (X![CAK]) => spawn(run(X env), exec))
```

6 Conclusion

We introduced MetaKlaim as a basic calculus for handling the temporal and the spatial dimensions of global computing. Our preliminary results demonstrate

that MetaKlaim can support programming of a variety of policies abstracting from the underlying system infrastructure. The meta programming primitives take care of describing how code manipulations is reflected in the underlying infrastructure. As the work outlined in this paper will progress, we believe to be able to demonstrate the full potentialities of MetaKlaim to address the peculiar aspects raised by global computing.

References

1. C. Calcagno, E. Moggi, T. Sheard, W. Taha. Closed types for safe imperative MetaML. submitted for publication, 2001.
2. L. Cardelli. Program Fragments, Linking, and Modularization. In *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1997.
3. L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computers Science*240(1), Elsevier 2000.
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. L. Cardelli, Abstractions for Mobile Computing, 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, 1999.
6. N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, ACM Press, 1989.
7. R. Davies. A temporal-logic approach to binding-time analysis. In *11*th *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.
8. R. Davies, F. Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 258–270, St. Petersburg Beach, 1996.
9. 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.
10. 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.
11. R. De Nicola, G. Ferrari, R. Pugliese. Programming Acess Control: The KLAIM Experience, In *Proc. CONCUR'2000*, *LNCS* 1877, 2000.
12. 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.
13. S. Drossopoulou. Towards an Abstract Model of Java Dynamic Linking and Verification. In *Proc. Types in Compilation*, 2000.
14. S. Drossopoulou, S. Eisenbach, D. Wragg. A fragment calculus - towards a model of separate compilation, linking and binary compatibility. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 147–156, Washington - Brussels - Tokyo, July 1999. IEEE.
15. G. Ferrari, E. Moggi, R. Pugliese. Global Types and Network Services. In Proc. *ConCoord: International Workshop on Concurrency and Coordination*, ENTCS 54 (Montanari and Sassone Eds), Elsevier Science, 2001.

16. A. Fuggetta, G. Picco, G. Vigna. Understanging Code Mobility, *IEEE Transactions on Software Engineering*, 24(5), IEEE Computer Society, 1998.
17. Fournet, C., Gonthier, G. Levy, J-J., Maranget, L., Remy, D. A Calculus of Mobile Agents, In Proc. CONCUR'96, *LNCS* 1119, Springer, 1996.
18. J-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. PhD. Thesis, Université Paris VII, 1972.
19. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, ACM Press, 1985.
20. R. Gray, D. Kotz, G. Cybenko and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, Mobile Agents and Security, *Lecture Notes in Computer Science*, Springer-Verlag, 1998.
21. M. Hennessy, J. Riely. Distributed Processes and Location Failures, *Theoretical Computers Science*, to appear, 2001.
22. M. Hennessy, J. Riely. Resource Access Control in Systems of Mobile Agents, *Information and Computation*, to appear, 2001.
23. M. Hicks and S. Weirich. A Calculus for Dynamic Loading, Technical report, MS-CIS-00-07, University of Pennsylvania, 2000, (available on line `http://www.cis.upenn.edu/ mwh/papers/loadcalc.pdf`).
24. S. Kamin, M. Callahan, L. Clausen. Lightweight and generative components II: Binary-level components. In *[31]*, pages 28–50, 2000.
25. The MetaML Home Page provides source code and online documentation `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.
26. E. Moggi, W. Taha, Z. Benaissa, T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.
27. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
28. Z. Qian, A. Goldberg, A. Coglio. A formal specification of JavaTM class loading. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, volume 35.10 of *ACM Sigplan Notices*, pages 325–336, N. Y., October 15–19 2000. ACM Press.
29. P.Sewell. Modules, Abstract Types and Distributed Versioning. In *Proc. ACM Symposium on Principles of Programming Languages*, ACM Press, 2001.
30. P. Sewell, P. Wojciechowski. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. *IEEE Concurrency*, 2000.
31. W. Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
32. J.-P. Talpin, P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
33. J.-P. Talpin, P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.
34. J. Vitek, G. Castagna. Towards a Calculus of Secure Mobile Computations. In *Workshop on Internet Programming Languages*, *LNCS* 1686, Springer, 1999.
35. A.K. Wright, M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.