Electronic Notes in Theoretical Computer Science 62 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume62.html` 17 pages

Higher-Order Types and Meta-Programming for Global Computing

G. Ferrari

Dipartimento di Informatica, Univ. Pisa, Italy

E. Moggi

Dipartimento di Informatica e Scienze dell'Informazione, Univ. Genova, Italy

R. Pugliese

Dipartimento di Sistemi e Informatica, Univ. Firenze, Italy

Abstract

MetaKlaim is a case study in modeling the spatial, temporal and security aspects necessary for global computing. MetaKlaim integrates MetaML (an extension of SML for multi-stage programming) and Klaim (a Kernel Language for Agents Interaction and Mobility), in order to allow interleaving of meta-programming activities (like assembly and linking of code fragments), security checks (like type-checking at administrative boundaries) and normal computational activities. The staging annotations of MetaML provide a fine-grain control of the temporal aspects, Klaim's primitives support location awareness, while the type system supports security through the use of *global* types (in combination with dynamic type-checking) and generic mobile code through the use of polymorphism (à la system F). The paper describes syntax, type system and operational semantics of MetaKlaim, states two type safety results, and exemplifies its use for describing mobile code applications.

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 behavior. They 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). Moreover, nomadic or mobile components may detach from a node and re-attach later on a different 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 [17] and [3] 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 [16], Klaim [8], the Distributed π-calculus [21], the Ambient calculus [4], the Seal calculus [34], and Nomadic Pict [31]. All these foundational models encompass a notion of location to reflect the idea of administrative domains: computations at a certain location are under the control of a specific authority. In other words, they focus on the *spatial* dimension (which is often referred to as *network awareness*) of WAN programming.

Another crucial aspect of WAN programming concerns the *temporal* dimension: the run-time system may interleave computational activities with meta-programming 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 they allow to reconfigure the application without having to restart it. Several papers have addressed the problem of formally understanding dynamic linking (and separate compilation) [2,27,24,10,23,30]; other approaches have tackled the problems of security in systems of mobile agents (see e.g. [20,12,13,9,22]).

Hence, the spatial and the temporal dimension of WAN programming have been studied at considerable depth but in *isolation*, and their interplay has not been properly formalized and understood, yet. This paper proposes a foundational model which integrates the spatial and 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 Klaim [8] and MetaML [26,1]. Klaim (Kernel Language for Agents Interaction and Mobility) is an experimental language, inspired by the Linda coordination model [18,5], specifically designed to model and to program WAN applications by exploiting mobility. MetaML supports most features of SML and meta-programming constructs. Meta-programming provides an ideal tool for describing customization and combination of software components, since the meta-programming constructs have the same status of the other programming constructs.

This paper is a follow-up of [14,15]. In [14] we addressed the problem of protecting hosts from attacks or misbehavior of mobile processes. We introduced HotKlaim (for Higher-order typed Klaim), a variant of Klaim that allows only mobility of *process abstractions*, i.e. processes parameterized with respect to the operations having a local meaning. The type system exploits *global values* and *types* to ensure that operations having a local meaning are used only locally. In [15] we describe a preliminary version of MetaKlaim, which has only simple types. In this paper we push forward the integration effort, and propose an integration of HotKlaim and MetaML that takes into account also the polymorphic and global types of HotKlaim.

The rest of the paper is organized as follows: Section 2 introduces the syntax of MetaKlaim, Section 3 gives the type system, Section 4 defines the operational semantics, Section 5 presents the type safety result (proofs are omitted), Section 6 gives a few examples of mobile code applications, and Section 7 draws some conclusions.

2 MetaKlaim

This section introduces MetaKlaim, a foundational calculus specifically designed to model both the spatial and temporal aspects of global computing. MetaKlaim integrates Klaim's primitives in MetaML: the staging annotations of MetaML provide a fine-grain control of the temporal aspects, while Klaim allows to model the spatial aspects of distributed concurrent applications, including mobility. Figure 1 summarizes the syntax of the calculus.

Definition 2.1 A **net** $N \in \mathsf{Net} \stackrel{\Delta}{=} \mu(\mathsf{L} \times (\mathsf{E}_0 + \{\mathsf{err}, \mathsf{dead}\}))$ is a multi-set of pairs consisting of a locality l and either a closed term e, or the tag err indicating that a process at l has crashed, or the tag dead indicating a dead computation or a placeholder for a tuple that has been removed. ■

Remark 2.2 [Types in Terms] In a global computing scenario most SW components available on the network are expected to be highly parameterized. Functional abstraction is not enough for expressing the desirable forms of parameterization. Also a limited form of polymorphism, like that supported by SML, appears inadequate (see Section 6). Therefore, in MetaKlaim we have included polymorphic types á la system F [19]. However, this design choice and the need for decidable type-checking at run-time (since the *input* primitive performs dynamic type-checking) imply that MetaKlaim programs have to include a lot of type information (at run-time). For a description of the $\Box$-types we refer to Remark 3.1.

An important topic for further research is to find type systems, that do not require heavy type annotations in terms, and at the same time guarantee

- Types $t \in \mathsf{T} ::= X \mid L \mid t_1 \to t_2 \mid (t_i | i \in m) \mid \langle t \rangle \mid \Box t \mid \forall X.t$

  Global Types $g \in \mathsf{G} ::= L \mid (g_i | i \in m) \mid \Box t$

- Contexts $\Gamma \in \mathsf{Ctx} ::= \emptyset \mid \Gamma, X^n \mid \Gamma, x{:}\, t^n{:}\, k$ where $k = \mathsf{l}, \mathsf{g}$

- Terms $e \in \mathsf{E} ::= x \mid l \mid nil \mid (mr_i | i \in n) \mid e_1\ e_2 \mid \mathsf{fix}\ x{:}\, t{:}\, k.e \mid (e_i | i \in m)$

  $\mid\ op\ e \mid \langle e \rangle \mid \tilde{}e \mid \%e \mid \Lambda X.e \mid e\{t\}$

  where l ranges over localities, nil is a deadlock computation, and op ranges over the set $\mathsf{Op} \stackrel{\Delta}{=} \{spawn, output, input, new, run\}$ of *local operations*

  Patterns $p \in \mathsf{P} ::= x!t{:}\, k \mid x = e \mid (p_i | i \in m)$

  Match Rules $mr ::= p \Rightarrow e$

Fig. 1. Syntax of types and terms

decidable type-checking (in linear time) and adequate expressiveness. ■

From Klaim we borrow the computational paradigm, which identifies *processes* as the basic 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 communicate asynchronously via TSs. The types of MetaKlaim include the types L of localities and $(t_i | i \in m)$ of tuples. The Klaim primitives take the form of polymorphic *local operations*:

- $spawn(e)$ activates a process in a parallel thread.
- $output(l, e)$ adds the value of e to the TS at l (*output* is non-blocking).
- $input(l, (p_i \Rightarrow e_i | i \in m))$ accesses the TS located at l. *input* checks each pattern p_i and looks in the TS at l for a matching value v. If such a v exists, it is removed from the TS, and the variables declared in the matching pattern p_j (i.e. those indicated by $x!t{:}\, k$) 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). Notice that *input* exploits 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 a process at l, and returns l.

Remark 2.3 [MetaKlaim versus Klaim] In Klaim there is a primitive $eval(l, e)$ for activating a process at a remote locality l. This primitive is used for process mobility, but it has not been included in MetaKlaim for the following reasons:

- *eval* relies on dynamic scoping (a potentially dangerous mechanism), which is not available in MetaKlaim, since in a functional setting one can use (the safer mechanism of) parameterization.

- with *eval* a node may activate a process on another node, but the target node has no control over the incoming process. This can be a source of security problems. In particular, Local Type Safety (see Theorem 5.2) fails, if *eval* is added.

In MetaKlaim process mobility occurs only by "mutual agreement", i.e. a (sending) node can *output* a process abstraction in any TS, but the abstraction may become an active process only if (a process at) another (receiving) node does *input* it. Remote communication between nodes, like that provided by Klaim, is essential to implement this form of process mobility. ■

From MetaML we borrow the types $\langle t\rangle$ for code with potentially unresolved links (represented by *dynamic* variables), the stratification of evaluation into levels (level 0 for normal evaluation, and level $n > 0$ for symbolic evaluation), and the following staging annotations:

- Brackets $\langle e\rangle$ constructs code representing the program fragment obtained by the symbolic evaluation of e, e.g. $\langle 2+x\rangle$ is a value of type $\langle \mathsf{nat}\rangle$ representing the fragment $2 + x$, where x is a *dynamic* variable.
- Escape $\tilde{}\, e$ returns the program fragment represented by e. During symbolic evaluation Escape is used for splicing program fragments into bigger programs, e.g. $\langle \lambda x.1 + \tilde{}\langle 2 + x\rangle\rangle$ evaluates to $\langle \lambda x.1 + 2 + x\rangle$.
- Cross-stage persistence $\%e$ allows to use the value of e at a higher level, e.g. $\langle \%(1+1)+x\rangle$ evaluates to $\langle \%2+x\rangle$. Notice that $\%(1+1)$ and $\tilde{}\langle 1+1\rangle$ have the same type, but their symbolic evaluation is different: the first evaluates to $\%2$, while the second evaluates to $1 + 1$.
- $run(e)$ executes the program represented by e, e.g. $run\langle 1 + 1\rangle$ evaluates to 2. To enhance security, *run* is considered a *local* operation.

Remark 2.4 [MetaKlaim versus MetaML] In MetaML (and more generally in multi-level languages) it is possible to evaluate under (dynamic) lambda. This feature is essential for allowing arbitrary interleaving of code generation and normal computation, but it may cause new forms of improper run-time behavior, that do not arise in traditional programming languages:

- execution of program fragments with unresolved links, e.g. the evaluation of $\langle \lambda x.\tilde{}(run\langle x\rangle;\ldots)\rangle$ will attempt to evaluate $run\langle x\rangle$, before the dynamic variable x gets bound to a value.
- extrusion of a value with free dynamic variables from the scope of the binding lambda, e.g. the evaluation of $\langle \lambda x.\tilde{}(output(l,\langle x\rangle);\ldots)\rangle$ will output $\langle x\rangle$ in the TS located at l, thus loosing the connection with the binding lambda.

There are two ways of addressing these problems.

Solution 1 is to design a type system which guarantees that well-typed programs do not exhibit improper behaviors. [1] adopts this approach, by introducing a type system with *closed types* (it introduces also a binder for dead code annotations, which is instrumental to the prove of type safety, and could be ignored when evaluating well-typed programs).

Solution 2 is to handle the problem dynamically, e.g. by stopping evaluation (of a thread), before an improper behavior occurs. This could be implemented by (dynamically) replacing all free variables in $\langle x\rangle$ with *nil*, before it is *run* or *output*. In a language supporting exception handling a better alternative is to replace these free variables with `raise unresolved_link`, instead of *nil*.

In a statically typed language the first solution is preferable, because it avoids run-time overheads. However, MetaKlaim has dynamic type-checking (because of the *input* primitive), therefore the above argument does not hold. In fact, a simpler dynamic type-checking algorithm may compensate the run-time overheads. Therefore, in MetaKlaim one has the following trade-off:

- solution 1 requires a more complex type system (e.g. with closed types and exhaustive pattern matching), but it is able to detect (at type-checking time) the possibility of improper run-time behavior
- solution 2 adopts a simpler type system, but it involves an overhead in implementing some operations (namely all local operations except *input*), which is linear is the size of the operand, and it may introduce deadlocked computations (as a consequence of replacing dynamic variables with *nil*).

We have chosen the second solution. ■

3 Type System

Figure 2 gives the type system for deriving judgments of the following forms

- $\Gamma \vdash$, i.e. Γ is a well-formed context
- $\Gamma \vdash_n t$, i.e. t is a well-formed type at level n
- $\Gamma \vdash_n e\colon t$, i.e. e is a well-formed term of type t at level n

A context Γ is a sequence of declarations of the form X^n for type variables and $x\colon t^n\colon k$ for term variables. A type variable X^n ranges over types t at level n, while a term variable $x\colon t^n\colon k$ ranges over *values* of type t at level n and of *kind* k. We consider also a subset G of T, whose elements are called global types. Semantically, a type t is global iff $t = \Box t$ (i.e. a term has type t iff it has type $\Box t$). In some typing rules we use the following derived judgement:

- $\Gamma \vdash_{n,\mathsf{g}} e\colon t$ means that $\Gamma \vdash$ and $\Gamma_{\mathsf{g}} \vdash_n e\colon t$ and e does not contain local operations *op*, where Γ_{g} is the context obtained from Γ by keeping only the declarations of the form X^m and $x\colon t^m\colon \mathsf{g}$.

Remark 3.1 [Levels, Kinds and $\Box$-types] **Levels** are typical of multi-level languages (like $\lambda^{\bigcirc}$ of [6]). In a dynamically typed multi-stage language, like MetaKlaim (see also [32]), type variables get bound at different stages of a

$$\frac{\Gamma \vdash}{\Gamma, X^n \vdash}\ X \text{ fresh} \qquad \frac{\Gamma \vdash_n t}{\Gamma, x{:}\, t^n{:}\, k \vdash}\ x \text{ fresh} \qquad \frac{\Gamma \vdash}{\Gamma \vdash_n X}\ X^m \in \Gamma \text{ and } m \leq n$$

$$\frac{\Gamma \vdash_n t_1 \quad \Gamma \vdash_n t_2}{\Gamma \vdash_n t_1 \to t_2} \qquad \frac{\Gamma \vdash}{\Gamma \vdash_n L} \qquad \frac{\Gamma \vdash_n t_i \quad i \in m}{\Gamma \vdash_n (t_i | i \in m)} \qquad \frac{\Gamma \vdash_{n+1} t}{\Gamma \vdash_n \langle t \rangle} \qquad \frac{\Gamma \vdash_n t}{\Gamma \vdash_n \Box t}$$

$$\frac{\Gamma, X^n \vdash_n t}{\Gamma \vdash_n \forall X.t} \qquad \frac{\Gamma \vdash}{\Gamma \vdash_n x{:}\, t}\ x{:}\, t^n{:}\, k \in \Gamma \qquad \frac{\Gamma \vdash}{\Gamma \vdash_n l{:}\, L} \qquad \frac{\Gamma \vdash_n t}{\Gamma \vdash_n \mathit{nil}{:}\, t}$$

$$\frac{\{\Gamma \vdash_n e(p_i){:}\, L \quad \Gamma, \Gamma^n(p_i) \vdash_n e_i{:}\, t_2 \mid i \in m\}}{\Gamma \vdash_n (p_i \Rightarrow e_i | i \in m){:}\, t_1 \to t_2}\ \forall i \in m. t(p_i) \equiv t_1$$

$$\frac{\Gamma \vdash_n e_1{:}\, t_1 \to t_2 \quad \Gamma \vdash_n e_2{:}\, t_1}{\Gamma \vdash_n e_1\ e_2{:}\, t_2} \qquad \frac{\Gamma, x{:}\, t^n{:}\, \mathsf{g} \vdash_{n,\mathsf{g}} e{:}\, t}{\Gamma \vdash_n \mathsf{fix}\ x{:}\, t{:}\, \mathsf{g}.e{:}\, t} \qquad \frac{\Gamma, x{:}\, t^n{:}\, \mathsf{l} \vdash_n e{:}\, t}{\Gamma \vdash_n \mathsf{fix}\ x{:}\, t{:}\, \mathsf{l}.e{:}\, t}$$

$$\frac{\{\Gamma \vdash_n e_i{:}\, t_i \mid i \in m\}}{\Gamma \vdash_n (e_i | i \in m){:}\, (t_i | i \in m)} \qquad \frac{\Gamma \vdash_n e{:}\, () \to t}{\Gamma \vdash_n \mathit{spawn}\ e{:}\, ()} \qquad \frac{\Gamma \vdash_n e{:}\, L \to t}{\Gamma \vdash_n \mathit{new}\ e{:}\, L}$$

$$\frac{\Gamma \vdash_n e{:}\, (L, \Box t_1 \to t_2)}{\Gamma \vdash_n \mathit{input}\ e{:}\, t_2} \qquad \frac{\Gamma \vdash_n e{:}\, (L, \Box t)}{\Gamma \vdash_n \mathit{output}\ e{:}\, ()} \qquad \frac{\Gamma \vdash_n e{:}\, \langle t \rangle \quad \Gamma \vdash_n t}{\Gamma \vdash_n \mathit{run}\ e{:}\, t}$$

$$\frac{\Gamma \vdash_{n+1} e{:}\, t}{\Gamma \vdash_n \langle e \rangle{:}\, \langle t \rangle} \qquad \frac{\Gamma \vdash_n e{:}\, \langle t \rangle}{\Gamma \vdash_{n+1} \text{\textasciitilde} e{:}\, t} \qquad \frac{\Gamma \vdash_n e{:}\, t}{\Gamma \vdash_{n+1} \% e{:}\, t} \qquad \frac{\Gamma, X^n \vdash_n e{:}\, t}{\Gamma \vdash_n \Lambda X.e{:}\, \forall X.t}$$

$$\frac{\Gamma \vdash_n e{:}\, \forall X.t_2 \quad \Gamma \vdash_n t_1}{\Gamma \vdash_n e\{t_1\}{:}\, t_2[X{:}= t_1]} \qquad \frac{\Gamma \vdash_{n,\mathsf{g}} e{:}\, t}{\Gamma \vdash_n e{:}\, \Box t} \qquad \frac{\Gamma \vdash_n e{:}\, g}{\Gamma \vdash_n e{:}\, \Box g} \qquad \frac{\Gamma \vdash_n e{:}\, \Box t}{\Gamma \vdash_n e{:}\, t}$$

Fig. 2. Type System

computation, and thus well-formedness is level dependent not only for terms, but also for types. In MetaKlaim we consider two **kinds** of terms: local terms, classified by kind l; global terms, classified by kind g, are those terms with no occurrences of *local* operations *op*. Type $\Box t$ is a *modality* (see $\lambda^{\Box}$ of [7]) related to the classification of terms, namely $\Box t$ classifies the global values of type t. Therefore, the following subset relation holds $\Box t \leq t$ (i.e. a term of type $\Box t$ has also type t). ■

We comment some of the typing rules:

- The rule for type variables supports a form of cross-stage persistence (as in [32]), namely an X declared at level m can be used at higher levels. The other rules for types are as expected.
- The rule for pattern-matching $(p_i \Rightarrow e_i | i \in m)$ enforces that all patterns have the same type, but it does not require that all cases are covered by at least

one pattern. The rule uses auxiliary notation (type $t(p)$, context $\Gamma^n(p)$ and the sequence $e(p)$ of terms) defined by induction on the structure of p:

p	$t(p)$	$\Gamma^n(p)$	$e(p)$
$x!t\colon \mathsf{g}$	$\Box t$	$x\colon t^n\colon \mathsf{g}$	$\emptyset$
$x!t\colon \mathsf{l}$	t	$x\colon t^n\colon \mathsf{l}$	$\emptyset$
$x = e$	$\Box L$	$x\colon L\colon \mathsf{g}$	e
$(p_i \mid i \in m)$	$(t(p_i) \mid i \in m)$	$\Gamma^n(p_0), \ldots, \Gamma^n(p_{m-1})$	$e(p_0), \ldots, e(p_{m-1})$

- The typing rule for $\mathsf{fix}\ x\colon t\colon \mathsf{g}.e$ and the introduction rules for $\Box t$ are the only ones that use the derived judgements $\Gamma \vdash_{n,\mathsf{g}} e\colon t$.
- Unlike [15], the typing for *run* (and the other local operations) does not use the closed type constructor of [1], since we have opted for a simpler type system (see Remark 2.4), at the expense of additional overhead in the implementation of the local operations.
- The last three rules for $\Box t$ are borrowed from [14]. They say that $\Box t$ is a *subset* of t, and that the two types coincide when t is a global type.

The type system enjoys the following property.

Proposition 3.2 (Substitution) *The following rules are admissible*

$$\frac{\Gamma_1 \vdash_m t \quad \Gamma_1, X^m, \Gamma_2 \vdash_n t'}{\Gamma_1, \Gamma_2[X\colon = t] \vdash_n t'[X\colon = t]} \qquad \frac{\Gamma_1 \vdash_m t \quad \Gamma_1, X^m, \Gamma_2 \vdash_n e'\colon t'}{\Gamma_1, \Gamma_2[X\colon = t] \vdash_n e'[X\colon = t]\colon t'[X\colon = t]}$$

$$\frac{\Gamma_1 \vdash_{m,\mathsf{g}} e\colon t \quad \Gamma_1, x\colon t^m\colon \mathsf{g}, \Gamma_2 \vdash_n e'\colon t'}{\Gamma_1, \Gamma_2 \vdash_n e'[x\colon = e]\colon t'} \qquad \frac{\Gamma_1 \vdash_m e\colon t \quad \Gamma_1, x\colon t^m\colon \mathsf{l}, \Gamma_2 \vdash_n e'\colon t'}{\Gamma_1, \Gamma_2 \vdash_n e'[x\colon = e]\colon t'}$$

4 Operational Semantics

The dynamics of a net is given by a relation $N \Longrightarrow N'$ defined in terms of transition relations $e \overset{a}{\longmapsto} e'$ (and $e \longmapsto \mathsf{dead} \mid \mathsf{err}$) for terms. The transitions relations are defined in terms of evaluation contexts (see [35]) and reductions $\Gamma, r^0 \xrightarrow{a} e'$ (and $\Gamma, r^0 \longrightarrow \mathsf{dead} \mid \mathsf{err}$) for actions and $r^1 \xrightarrow{1} e'$ (and $r^1 \longrightarrow \mathsf{err}$) for symbolic evaluation. Figure 3 summarizes the syntactic categories for the operational semantics. The net transition relation $\Longrightarrow$ is defined (in terms of $\longmapsto$) by the rules

$$\frac{e \longmapsto \mathsf{dead}}{N \uplus (l\colon e) \Longrightarrow N \uplus (l\colon \mathsf{dead})} \qquad \frac{e \longmapsto \mathsf{err}}{N \uplus (l\colon e) \Longrightarrow N \uplus (l\colon \mathsf{err})}$$

$$\frac{e \overset{\tau}{\longmapsto} e'}{N \uplus (l\colon e) \Longrightarrow N \uplus (l\colon e')} \qquad \frac{e \overset{i(v^0)@l_2}{\longmapsto} e'}{N \uplus (l_1\colon e) \uplus (l_2\colon v^0) \Longrightarrow N \uplus (l_1\colon e') \uplus (l_2\colon \mathsf{dead})}$$

- Values $v^n \in \mathsf{V}^n \subset \mathsf{E}$ at level $n \in \mathsf{N}$

$$v^0 ::= l \mid (vmr^0_i | i \in n) \mid (v^0_i | i \in m) \mid \langle v^1 \rangle \mid \Lambda X.e$$

$$v^{n+1} ::= x \mid l \mid nil \mid (vmr^{n+1}_i | i \in n) \mid v^{n+1}_1 v^{n+1}_2 \mid \mathsf{fix}\, x{:}\, t{:}\, k.v^{n+1} \mid (v^{n+1}_i | i \in m)$$
$$\mid \ op\ v^{n+1} \mid \langle v^{n+2} \rangle \mid \% v^n \mid \Lambda X.v^{n+1} \mid v^{n+1}\{t\}$$

$$v^{n+2} += \,\tilde{}\, v^{n+1}$$

Evaluated Patterns $vp^n \in \mathsf{VP}^{\cdot} ::= x!t{:}\, k \mid x = v^n \mid (vp^n_i | i \in m)$

Evaluated Match Rules $\quad vmr^0 ::= vp^0 {\Rightarrow} e$

$$vmr^{n+1} ::= vp^{n+1} {\Rightarrow} v^{n+1}$$

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

$$r^0 ::= x \mid nil \mid v^0_1 v^0_2 \mid \mathsf{fix}\, x{:}\, t{:}\, k.e \mid op\ v^0 \mid \tilde{}\, e \mid \% e \mid v^0\{t\}$$

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

- Evaluation Contexts $E^n_i \in \mathsf{EC}^n_i$ at level $n \in \mathsf{N}$ with hole at level $i \in \{0,1\}$

$$E^n_i ::= (\overline{vmr}^n, Ep^n_i {\Rightarrow} e, \overline{mr}) \mid E^n_i e \mid v^n E^n_i \mid (\overline{v}^n, E^n_i, \overline{e}) \mid op\ E^n_i \mid \langle E^{n+1}_i \rangle \mid E^n_i\{t\}$$

$$E^{n+1}_i += (\overline{vmr}^{n+1}, vp^{n+1} {\Rightarrow} E^{n+1}_i, \overline{mr}) \mid \mathsf{fix}\, x{:}\, t{:}\, k.E^{n+1}_i \mid \tilde{}\, E^n_i \mid \% E^n_i \mid \Lambda X.E^{n+1}_i$$

$$E^i_i += []$$

Evaluation Contexts for patterns $Ep^n_i ::= x = E^n_i \mid (\overline{vp}^n, Ep^n_i, \overline{p})$

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

Fig. 3. Values, redexes and evaluation contexts

$$\frac{e \overset{o(v^0)@l_2}{\longmapsto} e'}{N \uplus (l_1{:}\, e) \Longrightarrow N \uplus (l_1{:}\, e') \uplus (l_2{:}\, v^0)} \qquad \frac{e \overset{s(e_2)}{\longmapsto} e_1}{N \uplus (l{:}\, e) \Longrightarrow N \uplus (l{:}\, e_1) \uplus (l{:}\, e_2)}$$

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

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. The transition relation $\longmapsto$ is defined (in terms of $\longrightarrow$) by the rules

$$\frac{\Gamma^0(E^0_0), r^0 \xrightarrow{a} e'}{E^0_0[r^0] \overset{a}{\longmapsto} E^0_0[e']} \qquad \frac{r^1 \xrightarrow{1} e'}{E^0_1[r^1] \overset{\tau}{\longmapsto} E^0_1[e']}$$

$$\frac{\Gamma^0(E_0^0), r^0 \longrightarrow \mathsf{dead} \mid \mathsf{err}}{E_0^0[r^0] \longmapsto \mathsf{dead} \mid \mathsf{err}} \qquad \frac{r^1 \longrightarrow \mathsf{err}}{E_1^0[r^1] \longmapsto \mathsf{err}}$$

where $\Gamma^n(E_i^n)$ is the typing context for the hole in the evaluation context $E_i^n \in \mathsf{EC}_i^n$, this typing context is needed only by the reduction for *run*. Figure 4 defines the reduction $\longrightarrow$ and uses the following operations:

- Demotion $v^{n+1} \downarrow_{n,\Gamma} \in \mathsf{E}$ is defined by induction on $v^{n+1} \in \mathsf{V}^{n+1}$:

v^{n+1}	$v^{n+1} \downarrow_{n,\Gamma}$
x	nil if $x{:}\, t^m{:}\, k \in \Gamma$ x otherwise
$\langle v^{n+2}\rangle$	$\langle v^{n+2} \downarrow_{n+1,\Gamma}\rangle$
$\Lambda X.v^{n+1}$	$\Lambda X.v^{n+1} \downarrow_{n,\Gamma}$
$v^{n+1}\{t\}$	$v^{n+1} \downarrow_{n,\Gamma} \{t \downarrow_{n,\Gamma}\}$
${}^{\sim}v^n$	${}^{\sim}v^n \downarrow_{n-1,\Gamma}$ $(n > 0)$
$\mathsf{fix}\ x{:}\, t{:}\, k.v^{n+1}$	$\mathsf{fix}\ x{:}\, t \downarrow_{n,\Gamma}{:}\, k.v^{n+1} \downarrow_{n,\Gamma}$
$\%v^n$	$\%v^n \downarrow_{n-1,\Gamma}$ if $n > 0$ $\circ v^n$ otherwise

t	$t \downarrow_{n,\Gamma} \in \mathsf{T}$
X	$\forall X.X$ if $X^m \in \Gamma$ and $m > n$ X otherwise
$\langle t\rangle$	$\langle t \downarrow_{n+1,\Gamma}\rangle$
$\forall X.t$	$\forall X.t \downarrow_{n,\Gamma}$
vp^{n+1}	$vp^{n+1} \downarrow_{n,\Gamma} \in \mathsf{P}$
$x!t{:}\, k$	$x!t \downarrow_{n,\Gamma}{:}\, k$
$x = v^{n+1}$	$x = v^{n+1} \downarrow_{n,\Gamma}$
vmr^{n+1}	$vmr^{n+1} \downarrow_{n,\Gamma}$
$vp^{n+1} \Rightarrow v^{n+1}$	$vp^{n+1} \downarrow_{n,\Gamma} \Rightarrow v^{n+1} \downarrow_{n,\Gamma}$

$\circ v^n$ is the substitution instance of v^n with all free term variables replaced by *nil*.

In all other cases $\_ \downarrow_{n,\Gamma}$ commutes with the top level type and term construct.

- The functions $match(vp^0, v_0^0)$ and $match_t(vp^0, v_0^0)$ either $fail$ or return a $\rho{:}\, \mathsf{X} \overset{fin}{\to} \mathsf{V}_0^0$. They are defined by induction on $vp^0 \in \mathsf{VP}^0$, the base cases are:

vp^0	$match(vp^0, v_0^0)$	$match_t(vp^0, v_0^0)$	
$x!t{:}\, \mathsf{l}$	$x{:}= v_0^0$	$x{:}= v_0^0$	if $\emptyset \vdash_0 v_0^0{:}\, t$
$x!t{:}\, \mathsf{l}$	$x{:}= v_0^0$	$fail$	otherwise
$x!t{:}\, \mathsf{g}$	$x{:}= v_0^0$	$x{:}= v_0^0$	if $\emptyset \vdash_{0,\mathsf{g}} v_0^0{:}\, t$
$x!t{:}\, \mathsf{g}$	$x{:}= v_0^0$	$fail$	otherwise
$x = v^0$	$x{:}= v_0^0$	$x{:}= v_0^0$	if $v_0^0 \equiv v^0 \in \mathsf{L}$
$x = v^0$	$fail$	$fail$	otherwise

$match_t$ performs dynamic type-checking (and is used for *input* reduction), while *match* ignores types (and is used for function application).

- Closure $\bullet e \in \mathsf{E}_0$ is the substitution instance of e with all free term variables x replaced by *nil* and all free type variables X replaced by $\forall X.X$. $\bullet e$ is used for preventing scope extrusion (see Remark 2.4).

$$(vp^0_i{\Rightarrow}e_i|i\in n)\ v^0 \xrightarrow{\tau} e_j[\rho] \qquad \text{if } j\in n \text{ and } match(vp^0_j,v^0)=\rho \text{ and } \forall i<j.match(vp^0_i,v^0)=fail$$

$$(vp^0_i{\Rightarrow}e_i|i\in n)\ v^0 \longrightarrow \mathsf{dead} \qquad \text{if } \forall i\in n.match(vp^0_i,v^0)=fail$$

$$input\ (l,(vp^0_i{\Rightarrow}e_i|i\in n)) \xrightarrow{i(v^0)@l} e_j[\rho] \qquad \text{if } j\in n \text{ and } match_t(vp^0_j,v^0)=\rho \text{ and } \forall i<j.match_t(vp^0_i,v^0)=fail$$

$$nil \longrightarrow \mathsf{dead} \qquad \mathsf{fix}\ x{:}\,t{:}\,k.e \xrightarrow{\tau} e[x{:}=\mathsf{fix}\ x{:}\,t{:}\,k.e]$$

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

$$\Gamma, run\ \langle v^1\rangle \xrightarrow{\tau} v^1\downarrow_{0,\Gamma} \qquad (\Lambda X.e)\{t\} \xrightarrow{\tau} e[X{:}=t]$$

$$r^0 \longrightarrow \mathsf{err} \text{ in all other cases except when } r^0\equiv input\ (l,(vp^0_i{\Rightarrow}e_i|i\in n))$$

$$\tilde{}\langle v^1\rangle \xrightarrow{1} v^1 \qquad r^1 \longrightarrow \mathsf{err} \text{ in all other cases}$$

Fig. 4. Reductions for actions and symbolic evaluation

5 Type Safety

To state the type safety property we introduce two notions of well-formed net: one is a global property, the other is relative to a subset L of nodes.

Definition 5.1 [Well-formed Net] A net N is well-formed $\stackrel{\Delta}{\Longleftrightarrow}$ $(l{:}\,\mathsf{err})\notin N$ when $l\in L(N)$, and for every $(l{:}\,e)\in N$ exists t s.t. $\emptyset\vdash_0 e{:}\,t$.

A net N is well-formed w.r.t. $L\subseteq L(N)$ $\stackrel{\Delta}{\Longleftrightarrow}$ $(l{:}\,\mathsf{err})\notin N$ when $l\in L$, and for every $(l{:}\,e)\in N$ with $l\in L$ and $e\notin \mathsf{V}^0$ exists t s.t. $\emptyset\vdash_0 e{:}\,t$. ■

In the definition of well-formed net w.r.t. L nothing is said about evaluated tuples v^0 located in L. In fact, a process external to L may insert an ill-typed tuple in a node of L.

Theorem 5.2 (Type Safety) *If $N \Longrightarrow N'$, then*

- *(Global) N well-formed implies N' well-formed*
- *(Local) N well-formed w.r.t. L implies N' well-formed w.r.t. L*

Remark 5.3 The local safety property is enforced by two features of MetaKlaim: the dynamic type-checking performed by the input operation, which prevents ill-typed terms to pollute a well-typed process (since $match_t$ fails on ill-typed tuples); the absence of Klaim's *eval* primitive, which would allow processes external to L to activate ill-typed processes located in L. ■

6 Examples

It has been widely acknowledged that *mobility* will provide the right abstraction to design and implement WAN applications. In particular, the usefulness of mobility emerges when developing both applications for devices with intermittent access to the network, and network services having different access policies. We refer to [29,17] for a detailed discussion on the mobility paradigm. Another crucial issue of WAN applications concerns component programming. Components available on the network are expected to be highly parameterized, in order to accommodate a multiplicity of applications 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 [25] give several examples of components for generating object-code (for instance in Java). These components are described as higher-order macros in a *functional* meta-language with Bracket and Escape constructs similar to those of MetaML. In this section, we exemplify the use of MetaKlaim to program WAN applications. In particular, we discuss two specific examples. The first example (nomadic data collector) addresses the issues of mobility. The second example (dynamic linker and loader) discusses generative component programming.

In the rest of this section, we will freely use features typical of functional languages (such as local declarations, datatypes, etc.). Moreover, for typesetting reasons, we write `[]_` instead of $\Box\_$, and `V X._` instead of $\forall X.\_$

6.1 Nomadic data collector

Consider the following scenario. A certain user requires to assemble information on a piece of data (e.g. the price of certain devices). Part of the behavior of the user's application strictly depends on this information. However, there are some activities which are independent of it. The user's application can be structured to exploit the mobility paradigm: a mobile component can dynamically travel among hosts of the net looking for the required information. Here, for simplicity, we assume that each node of the 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 make use of the following types:

```
L            (* localities *)
Key  = ...   (* authorization keys *)
Data = ...
(* polymorphic types of local operations input, output, spawn *)
I = V X. V Y. (L,[]X->Y) -> Y
O = V X. (L,[]X) -> ()
S = V X. (() -> X) -> ()
(* polymorphic types of meta-operations for input, output, spawn *)
```

```
MI = V X. V Y. (<L>,<[]X->Y>) -> <Y>
MO = V X. (<L>,<[]X>) -> <()>
MS = V X. (<() -> X>) -> <()>
(* code abstractions with static security checks *)
MEnvK = (<L>,Key->MI,Key->MO,MS)
CAK = MEnvK -> <()>
```

Polymorphic types of meta operations exploit code types, hence meta operations are able to insert the code fragments of the operations provided locally into larger programs. The type of code abstractions (e.g. the type of mobile code) are parameterized with respect to the locality (where the code will be executed) and the meta-operations. In other words, the type of code abstractions can be intuitively interpreted as the network environment of the code. This environment must be fed with the information about the current locations and its local operations. We want to emphasize the fact that the meta-operations for communication require an authorization key as parameter. In such a way, depending on the value of the key `k`, the meta-operation `in' k` could generate an actual `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.

We now discuss the main module of our mobile application: the nomadic data collector. The code abstraction `pca(k,i,u)` is the mobile code which retrieves the required information on the distributed database. The parameter `k` is an authorization key, `i` is a search key, and `u` is the locality (the address of the end-system) where all (the remote) data associated to `i` should be collected. The behavior of the mobile code `pca(k,i,u)` is rather intuitive. After being activated, `pca(k,i,u)` spawns a process that perform a *local query* (here the query removes data which are associated in the local database to the search key `i`). Then the mobile code forwards the result of the query to the tuple space located at `u`, and sends copies of itself (i.e. of `pca(k,i,u)`) to localities that may contain data associated to `i`. Notice that in the code abstraction `pca(k,i,u)` the cross-stage persistence allows one to use the parameters `i` and `u` at a higher level.

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

The code abstraction `pca(k,i,u)` is instantiated and activated by process `execute`. This process fetches code abstractions of type `CAK` from the local tuple space, instantiates them by providing the locality of the node and the meta-operations, and finally activates the specialized code.

```
execute (self:L:g, env:MEnvK:l) :():l =
  fix exec:():l.
      input (self, X!CAK:g => spawn (() => run(X env)) ; exec)
```

6.2 Dynamic Linking and Loading

In global computing programming one important issue is the ability to control the loading policy of components. For instance, the Java Virtual Machine supports dynamic linking and loading of classes [28,11]. In some cases (localities with good connectivity or trusted 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 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 *)
Key  = ...    (* authorization keys *)
Name = ...    (* service names *)
Linker=Name -> Key      (* linker *)
MLinker=Name -> <Key>   (* meta-linkers *)
(* parameterized application code *)
CApp=MLinker -> <()>
```

The process `execute` fetches application code from the local tuple place, generates code by passing a meta-linker, and then spawns a process that executes the generated code

```
Mexecute (self:L:g, mlinker:MLinker:l):():l =
  fix exec:():l.
      input(self, x!CApp:g => 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.

7 Conclusions

MetaKlaim is a calculus that combines a variety of features, coming from languages for mobility, coordination and staging. In global computing mobility seems a fundamental aspect, coordination is important for control and security, staging is important for assembling components and generating specialized code prior execution. To the best of our knowledge, MetaKlaim is the first language that integrates multi-stage programming features with formalisms for mobile processes.

Acknowledgements. We thank the referees for their useful comments.

References

[1] Calcagno, C., E. Moggi and T. Sheard, *Closed types for safe imperative MetaML* (2001), submitted for publication.

[2] Cardelli, L., *Program fragments, linking, and modularization*, in: *Proc. of ACM Symposium on Principles of Programming Languages* (1997).

[3] Cardelli, L., *Abstractions for Mobile Computation*, in: J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, Springer-Verlag, 1999 pp. 51–94.

[4] Cardelli, L. and A. Gordon, *Mobile ambients*, Theoretical Computer Science **240** (2000).

[5] Carriero, N. and D. Gelernter, *Linda in Context*, Comm. of the ACM **32** (1989), pp. 444–458.

[6] Davies, R., *A temporal-logic approach to binding-time analysis*, in: *the Symposium on Logic in Computer Science (LICS '96)*, IEEE Computer Society Press, New Brunswick, 1996, pp. 184–195.

[7] Davies, R. and F. Pfenning, *A modal analysis of staged computation*, in: *the Symposium on Principles of Programming Languages (POPL '96)*, St. Petersburg Beach, 1996, pp. 258–270.

[8] De Nicola, R., G. Ferrari and R. Pugliese, Klaim*: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 315–330.

[9] De Nicola, R., G. Ferrari, R. Pugliese and B. Venneri, *Types for Access Control*, Theoretical Computer Science **240** (2000), pp. 215–254.

[10] Drossopoulou, S., *Towards an abstract model of java dynamic linking and verification*, in: *Proc. of Types in Compilation*, 2000.

[11] Drossopoulou, S., S. Eisenbach and D. Wragg, *A fragment calculus - towards a model of separate compilation, linking and binary compatibility*, in: *Proc of the 14th Symposium on Logic in Computer Science (LICS'99)* (1999), pp. 147–156.

[12] Erlingsson, U. and F. Schneider, *SASI Enforcement of Security Policies: A Retrospective*, Technical Report TR99-1758, Cornell University (1999).

[13] Evans, D. and A. Twynman, *Flexible policy-directed code safety*, in: *Proc. of the IEEE Symposium on Security and Privacy* (1999).

[14] Ferrari, G., E. Moggi and R. Pugliese, *Global types and network services*, in: U. Montanari and V. Sassone, editors, *Proc of ConCoord: International Workshop on Concurrency and Coordination*, number 54 in ENTCS (2001).

[15] Ferrari, G., E. Moggi and R. Pugliese, MetaKlaim*: Metaprogramming for Global Computing*, in: V. Taha, editor, *Proc of Semantics, Applications, and Implementation of Program Generation (SAIG01)*, number 2196 in LNCS (2001), pp. 183–198.

[16] Fournet, C., G. Gonthier, J. J. Levy, L. Maranget and D. Remy, *A Calculus of Mobile Agents*, in: U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, LNCS **1119** (1996), pp. 406–421.

[17] Fuggetta, A., G. Picco and G. Vigna, *Understanging Code Mobility*, IEEE Transactions on Software Engineering **24** (1998).

[18] Gelernter, D., *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[19] Girard, J., "Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur," Thèse de doctorat d'etat, University of Paris VII (1972).

[20] Gray, R., 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*, LNCS, Springer-Verlag, 1998 .

[21] Hennessy, M. and J. Riely, *Distributed Processes and Location Failures*, Theoretical Computer Science (1997), to appear.

[22] Hennessy, M. and J. Riely, *Resource Access Control in Systems of Mobile Agents*, in: U. Nestmann and B. C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, ENTCS **16.3** (1998), pp. 3–17, to appear in Information and Computation.

[23] Hicks, M. and S. Weirich, *A Calculus for Dynamic Loading*, Technical Report MS-CIS-00-07, University of Pennsylvania (1994).
URL `http://www.cis.upenn.edu/~mwh/papers/loadcalc.pdf`

[24] Hicks, M., S. Weirich and K. Crary, *Safe and flexible dynamic linking of native code*, in: R. Harper, editor, *Proc. of Types in Compilation: Third International Workshop, TIC 2000*, LNCS **2071** (2000), pp. 147–176.

[25] Kamin, S., M. Callahan and L. Clausen, *Lightweight and generative components II: Binary-level components*, in: *[33]*, 2000, pp. 28–50.

[26] *The MetaML Home Page* (2000), provides source code and documentation online at `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.

[27] Morrisett, G. and N. Glew, *Type-safe linking and modular assembly language*, in: *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999), pp. 250–261.

[28] Qian, Z., A. Goldberg and A. Coglio, *A formal specification of JavaTM class loading*, , **35.10 ACM Sigplan Notices** (2000), pp. 325–336.

[29] Roman, G.-C., G. P. Picco and A. L. Murphy, *Software engineering for mobility*, in: *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)* (2000), pp. 241–260.

[30] Sewell, P., *Modules, Abstract Types and Dited Versioning*, in: *Proc. of ACM Symposium on Principles of Programming Languages* (2001).

[31] Sewell, P. and P. Wojciechowski, *Nomadic Pict: Language and Infrastructure Design for Mobile Agents*, IEEE Concurrency (2000).

[32] Shields, M., T. Sheard and S. L. Peyton Jones, *Dynamic typing through staged type inference*, in: *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 1998, pp. 289–302.

[33] Taha, W., editor, “Semantics, Applications, and Implementation of Program Generation,” Lecture Notes in Computer Science **1924**, Springer-Verlag, Montréal, 2000.

[34] Vitek, J. and G. Castagna, *Towards a calculus of secure mobile computations*, in: *Proc. Workshop on Internet Programming Languages* (1999).

[35] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.