Under consideration for publication in Math. Struct. in Comp. Science

MetaKlaim: A Type Safe Multi-stage Language for Global Computing

GIANLUIGI FERRARI[1†] and EUGENIO MOGGI[2‡] and ROSARIO PUGLIESE[3§]

[1] *Dipartimento di Informatica, Università di Pisa, Italy.*
[2] *Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy.*
[3] *Dipartimento di Sistemi e Informatica, Università di Firenze, Italy.*

Received 16 January 2004

This paper describes the design and the semantics of MetaKlaim, an higher order distributed process calculus equipped with staging mechanisms. MetaKlaim integrates MetaML (an extension of SML for multi-stage programming) and Klaim (a Kernel Language for Agents Interaction and Mobility), to permit interleaving of meta-programming activities (like assembly and linking of code fragments), dynamic checking of security policies at administrative boundaries and "traditional" computational activities on a wide area network (like remote communication and code mobility). MetaKlaim exploits a powerful type system (including polymorphic types á la system F) to deal with highly parameterized mobile components and to dynamically enforce security policies: types are metadata which are extracted from code at run-time and are used to express trustiness guarantees. The dynamic type checking ensures that the trustiness guarantees of wide are network applications are maintained whenever computations interoperate with potentially untrusted components.

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. Differently from traditional middle-wares for distributed programming, the structure of the underlying network is made manifest to programmers of WAN applications. In general, 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 and have to deal with the unpredictable changes over time of the network environment (changes due to the unavailability of network connectivity, lack of services, node failures, network reconfiguration,

† Supported by MIUR project NAPOLI and EU project PROFUNDIS IST-2001-33100.
‡ Supported by MIUR project NAPOLI and EU project DART IST-2001-33477.
§ Supported by MIUR project NAPOLI and EU project AGILE IST-2001-32747.

and so on). Moreover, nomadic or mobile components must be designed to support heterogeneity and interoperability because they may detach from a node and re-attach later on a different node. We refer to (FPV98) and (Car99) for comprehensive analysis of issues related to WAN applications.

The problems associated with the development of WAN applications have prompted the study of the foundations of programming languages with advanced features including mechanisms for code and 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 (FGL$^+$96), KLAIM (DFP98), the Distributed π-calculus (HR01), the Ambient calculus (CG00a), the Seal calculus (VC99), and Nomadic Pict (SW00). 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.

In a WAN setting no central authority can define and enforce policies which regulate accesses to network resources. Moreover, components of applications should be designed to be executed and interoperate with potentially malicious components. The advent of *safe* programming languages such as Java and C♯ has led to the definition of strong type systems which can be effectively exploited to rule out a variety of security bugs. Notice that the use of a typed intermediate code has come into prominence in the last few years (e.g. the Java Bytecode or the Microsoft Intermediate Language). However, static type checking cannot detect all potential security holes. Current run-time environments exploit some mechanisms (e.g. see (SMH00; Sch00; FBF99)), like reference monitors and certifying compilers, to *dynamically check* security properties which cannot be enforced either statically or at linking time. Dynamic enforcing of security checks increases the security level of WAN applications, because it permits to identify those portions of applications code that are potentially untrusted, and can support the revocation of previously granted permissions to partially trusted code. Moreover, components of WAN applications are often developed and maintained by different providers/principals and may be downloaded and linked together "on demand". Hence, the run-time system may interleave computational activities with meta-programming activities, such as dynamic linking, assembling and customization of components, that permit to reconfigure the application without having to restart it. In fact, the interest towards formally understanding dynamic linking (and separate compilation) is witnessed by several papers that have recently tackled the problem (Car97a; MG99; HWC00; Dro00; HW00; Sew01). To sum up, dynamic enforcement of security properties together with dynamic assembling and customization of components make the *temporal* dimension of WAN programming.

The spatial and the temporal dimensions 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 features of the problem in a calculus having primitives for programming processes which may migrate among nodes, and primitives which support fine-grain control on

dynamic linking of components and dynamic checking of security policies. Our calculus, called MetaKlaim, builds on Klaim (DFP98; KHP98; DFPV00) and MetaML (TS97; She99; TS00; MHP00). Klaim (Kernel Language for Agents Interaction and Mobility) is an experimental language, inspired by the Linda coordination model (Gel85; CG89), specifically designed to model and to program WAN applications by exploiting distribution and mobility. MetaML supports staging constructs for meta-programming and most features of SML. It is ideal for describing customization and combination of software components, since the staging constructs have the same status of the other programming constructs.

MetaKlaim takes the form of an higher-order distributed process calculus where staging handles naturally typed code. The calculus is designed around the following ingredients:

— Localities and code mobility to deal with the spatial dimension of WAN programming;
— Polymorphic types á la system F to deal with highly parameterized mobile components;
— Types, namely metadata extracted at run-time from potential untrusted code, to dynamically enforce security policies;
— Staging and meta-programming constructs (á la MetaML) to link, specialize, adapt, run and reconfigure mobile components by taking advantage of run-time type information.

In this paper we introduce the operational semantics of MetaKlaim. To our knowledge, this is the first semantics for a general-purpose higher order distributed process calculus with staging constructs. The operational semantics performs dynamic type checking of untrusted code, thus the trustness guarantees of wide are network applications are maintained, even when they interoperate with potentially untrusted components. Moreover, the type system of MetaKlaim and the dynamic type checking can ensure *local* Type Safety, i.e. type safety of just that part of the net we want to control.

The rest of the paper is organized as follows: Section 2 gives further motivations for our work; Section 3 presents the syntax of MetaKlaim and discusses the main linguistic design choices; Section 4 and 5 introduce a type system and define the operational semantics for MetaKlaim; Section 6 states and demonstrates the type safety result; Section 7 gives a few examples of distributed and mobile code applications; Section 8 presents some comparisons with related work; finally, Section 9 draws some conclusions and discusses directions for further work.

2. Further Motivations

Current software technologies emphasize the notion of *components* as the key idiom to control the design and the development of applications. Ideally, programmers should design and build applications by combining and integrating together (pre-existing) components. To support this simple idea, programming languages should provide mechanisms to *link and specialize* components. In other words, components are assumed to be *generic* and *pluggable* to other components to achieve the required functionalities.

Modern operating systems and programming languages (such as Java) include *dynamic linking* mechanisms as a fundamental part of their run time environment. COM (Cor01) and Java Beans (Mic02) support component updating: run-time type checking is used to determine what versions of components are available.

Components often embody facilities to specialize their structure and generate efficient code once the parameters of the components have been provided. These components are called *generative* (EC00). An illustrative example is given by C++ template mechanisms and template metaprogramming (MS96). Multi-stage programming languages have been proposed for writing generative components. For instance, (KCC00) presents several examples of components, described as higher-order macros in a functional meta-language similar to MetaML.

The execution cycle of component-based programming (that is characterized by the ability of integrating components into applications) consists of

1 Finding the required components;
2 Linking and specializing components;
3 Running the application assembled from components.

The advent of network technologies introduces new phenomena: components are available on the net and are managed (and provided) by different authorities. The use of components in a WAN environment raises a number of interesting issues. First, given the heterogeneity of the environment net components should be highly portable: components could be used anywhere but require some services to behave properly (i.e. services are used to adapt components to a variety of infrastructures). Functional abstraction is not enough for expressing the desirable forms of parametrization. Also a limited form of polymorphism, like that supported by SML, appears inadequate. Second, security should be ensured: components downloaded from different authorities have different security requirements, and they should be executed within different run-time environments. Third, dynamic adaptability should be ensured: WAN applications are highly dynamic and can reconfigure their structure and their components at run-time to respond to dynamic changes of the network environment.

Thus, in the case of WAN programming, the execution cycle of components includes additional steps and becomes:

1 Downloading generic components;
2 Adapting components to the local infrastructure and the local execution environment;
3 Fixing the loading and specialization policies according to local requirements;
4 Monitoring the execution of the assembled application;
5 Reconfiguring the application and its policies whenever the network environment changes.

The refined execution cycle (we will call it the *network cycle*), also applies to *nomadic* (mobile) applications: it suffices to substitute the *Download* step with a *MoveTo* step. This is important because it has been widely acknowledged that *mobility* (FPV98; RPM00) provides a suitable 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.

Current technologies provide solutions only to some of the issues discussed above. For instance, in the Java programming language heterogeneity is handled through bytecode interpretation. Permissions, grants and stack inspection handles dynamic check of possibly untrusted code. C♯ *generics* account for highly parameterized generic components (KS01). The .NET architecture supplies a programming technology embodying general facilities for handling heterogeneity and orchestration of WEB services.

3. MetaKlaim

This section introduces MetaKlaim, a foundational multi-stage calculus specifically designed to model both the spatial and temporal aspects of global computing. MetaKlaim extends system F (Gir72; Rey74) with primitives from Klaim and MetaML: Klaim's primitives permit to model the spatial aspects of distributed concurrent applications, including code mobility, while the staging annotations of MetaML provide a fine-grain control of the temporal aspects.

Notation 3.1 (Notations and Conventions used throughout the paper).

— m, n range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} | i < m\}$ of its predecessors.
— Syntactic equivalence, written $\equiv$, is α-conversion. $\mathrm{FV}(e)$ is the set of free variables in e. If E is a set of syntactic entities, then E_0 indicates the set of entities in E without free variables.
— $\overline{e}$ ranges over finite sequences of e. $|\overline{e}|$ is the number of elements in the sequence $\overline{e}$. $\overline{e}_1, \overline{e}_2$ denotes the concatenation of the sequences $\overline{e}_1$ and $\overline{e}_2$ (and similarly for sequences Γ_1 and Γ_2 of declarations). $\overline{e} : t$ is a shorthand for $e_i : t$ for each e_i in the sequence $\overline{e}$.
— ρ ranges over substitutions, i.e. functions (with finite domain) mapping variables to terms (or types). $\emptyset$ is the empty substitution, $x := e$ is the substitution mapping x to e, and ρ_1, ρ_2 denotes the union of two substitutions (with disjoint domains). $e[\rho]$ is the result (modulo α-conversion) of applying the substitution ρ to e.
— $\mu(A)$ is the set of multisets with elements in A, and $\uplus$ is multiset union.
— Given a BNF $e := P_1 \mid \ldots \mid P_m$, we write $e+ = P_{m+1} \mid \ldots \mid P_{m+n}$ as a shorthand for the extended BNF $e := P_1 \mid \ldots \mid P_{m+n}$. □

From Klaim (DFP98; KHP98; DFPV00) 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 and $(t_i | i \in m)$ of localities and tuples, but not a type of processes, because process actions can be performed by terms of any type. In MetaKlaim the primitives of Klaim take the following form:

— $spawn(e)$ activates a process (obtained from e) in a parallel thread.

— $new(e)$ creates a new locality l, activates a process (obtained from e) at l, and returns l.
— $output(l, e)$ 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 for gathering data. The *input* operation 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 $x!t$ declared in the matching pattern p_j 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).

Remark 3.2. In KLAIM there is a primitive $eval(l, e)$ for activating a process at a remote locality l. This primitive is used for asynchronous 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) parametrization.
— 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 6.2) fails, if *eval* would be 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 becomes an active process only if (a process at) another (receiving) node *input* it. Higher-order remote communication between nodes, like that provided by KLAIM, is essential to implement this form of mobility. □

From METAML (TS97; She99; TS00; MHP00) we borrow the types $\langle t \rangle$ for code with potentially unresolved links (represented by *dynamic* variables), the stratification into levels of declarations (level $n > 0$ for dynamic variables) and evaluation (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 code 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* $\%e$ permits 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 code e, e.g. $run\langle 1 + 1 \rangle$ evaluates to 2.

Remark 3.3. In METAML it is possible to evaluate under (dynamic) lambda. This feature is essential for allowing arbitrary interleaving of code generation and normal

— Types $t \in \mathsf{T} ::= \quad X \mid L \mid t_1 \to t_2 \mid (t_i | i \in m) \mid \langle t \rangle \mid \forall X.t \mid U \Rightarrow t$
— Contexts $\Gamma \in \mathsf{Ctx} ::= \emptyset \mid \Gamma, X^n \mid \Gamma, x : t^n$
— Terms $e \in \mathsf{E} ::= \quad x \mid l \mid \lambda x : t.e \mid e_1\ e_2 \mid \mathsf{fix}\ x : t.e \mid (e_i | i \in m) \mid \pi_j\ e \mid op\ e$
$\quad \mid \quad \langle e \rangle \mid \tilde{}e \mid \%e \mid \Lambda X.e \mid e\{t\} \mid (mr_i | i \in m)$ with $m > 0$
Patterns $p \in \mathsf{P} ::= \quad x!t \mid x = e \mid (p_i | i \in m)$
Match Rules $mr ::= \quad p \Rightarrow e$

Fig. 1. Syntax of types and terms

computation, but it may cause new forms of improper run-time behavior, that do not arise in traditional programming languages:

— execution of code 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.

In a statically typed language these improper behaviors can be prevented by a more sophisticated type system, e.g. (CMS02) exploits *closed types*. For a language with dynamic type-checking, like MetaKlaim, we prefer to keep the type system simple (the cost of type checking is linear in the size of the term). The trade-offs are a run-time overhead on local operations (linear in the size of their operands) for checking the absence of unresolved links or scope extrusion, and run-time exceptions *exn* raised when a check detects a problem. □

Figure 1 summarizes the syntax of MetaKlaim, which uses the following primitives categories

— a numerable set XT of *type variables*, ranged over by $X, \ldots$;
— a numerable set X of *term variables*, ranged over by $x, \ldots$;
— a numerable set L of *localities*, ranged over by $l, \ldots$;
— a finite set $\mathsf{Op} = \{spawn, new, output, input, run\}$ of *local operations*, ranged over by *op*.

The syntax of MetaKlaim can be explained in terms of system F, Klaim and MetaML.

— From system F we borrow functional types $t_1 \to t_2$, abstraction $\lambda x : t.e$ and application $e_1\ e_2$, and polymorphic types $\forall X.t$, type abstraction $\Lambda X.e$ and instantiation $e\{t\}$.
— From Klaim we borrow localities l of type L, tuples $(e_i | i \in m)$ of type $(t_i | i \in m)$, and the construct $(p_i \Rightarrow e_i | i \in m)$ of type $U \Rightarrow t$, which performs pattern matching and dynamic type-checking on *untrusted* values deposited in tuple spaces (in Klaim this construct is bundled with the *input* primitive); the primitives *spawn*, *new*, *output* and *input* are among the local operations Op.
— From MetaML we borrow code types $\langle t \rangle$, and the staging annotations brackets $\langle e \rangle$, escape $\tilde{}e$, and cross-stage persistence $\%e$; *run* is among the local operations Op.
— Finally, we have recursive definitions $\mathsf{fix}\ x : t.e$ and projections $\pi_j\ e$.

In MetaKlaim, we perform a dynamic type check, when we input an untrusted value from a tuple space, in order to ensure some trustiness guarantees. The type system of MetaKlaim is relatively simple, and the guarantees we can express are limited. For instance, we cannot express constrains on the computational effects of a term, such as the ability to spawn new threads or to perform input/output. We circumvent this limitation of the type system by allowing only the input of *global* values.

Definition 3.4. A term $e \in \mathsf{E}_0$ is **global** $\stackrel{\Delta}{\Longleftrightarrow}$ it has no occurrences of *local* operations $op \in \mathsf{Op}$.

Thus the only way we can turn a global value v into a process (interacting with its environment) is by passing some local operations (possibly in customized form), in other words v must be a higher-order abstraction representing processes parameterized w.r.t. customized local operations. Even if we improve the expressiveness of type system by adding effects, there is still a need to consider processes parameterized w.r.t. customized local operations, and this parameterization will require also effect polymorphism (besides type polymorphism).

Remark 3.5. The use of dynamic type dispatching in a distributed polymorphic programming language has been strongly advocated in (Dug99). For simplicity, we have chosen not to include dynamic type dispatching in MetaKlaim, but it would be a very appropriate extension. However, one may wonder whether $input(x!t\Rightarrow e)$ of MetaKlaim is *semantically equivalent* to $\mathsf{typecase}\ \_\ \mathsf{of}\ (x:t)e$ of (ACPP91; ACPR95). In fact, they are different! To simplify the comparison we consider a type U of untrusted values, and replace the *input* primitive with a construct $\mathsf{check}\ \_\ \mathsf{against}\ (x!t)e$.

— The type U of untrusted values has the following introduction and elimination rules

$$\frac{\Gamma \vdash}{\Gamma \vdash u(e) : U} \qquad \frac{\Gamma \vdash v : U \qquad \Gamma, x : t \vdash e : t}{\Gamma \vdash \mathsf{check}\ v\ \mathsf{against}\ (x:t)e : t}$$

the reduction semantics is $\mathsf{check}\ u(v)\ \mathsf{against}\ (x:t)e \longrightarrow e[x := v]$ provided $\emptyset \vdash v : t$, thus at run-time we have to check that v has type t (in the empty context).

— In (ACPP91; ACPR95) the type D of dynamics has similar introduction and elimination rules (provided we do not consider pattern variables in types)

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash d(e:t) : D} \qquad \frac{\Gamma \vdash v : D \qquad \Gamma, x : t \vdash e : t}{\Gamma \vdash \mathsf{typecase}\ v\ \mathsf{of}\ (x:t)e : t}$$

the reduction semantics is $\mathsf{typecase}\ d(v:t')\ \mathsf{of}\ (x:t)e \longrightarrow e[x := v]$ provided $t' \equiv t$, thus at run-time we only need to check equality of types.

Therefore, the two mechanisms accomplish different useful tasks. For instance, if we have an untrusted dynamic value $u(d(v:t))$, we must first check that $d(v:t) : D$ (or equivalently that $v : t$), and only then we can compare t with other types to decide how to use v safely. □

4. Type System

The type system derives judgments of the following forms

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash}{\Gamma, X^n \vdash}\ X \text{ fresh} \qquad \frac{\Gamma \vdash_n t}{\Gamma, x : t^n \vdash}\ x \text{ fresh}$$

$$X\ \frac{\Gamma \vdash}{\Gamma \vdash_n X}\ X^m \in \Gamma \text{ and } m \leq n \qquad L\ \frac{\Gamma \vdash}{\Gamma \vdash_n L} \qquad \to\ \frac{\Gamma \vdash_n t_1 \quad \Gamma \vdash_n t_2}{\Gamma \vdash_n t_1 \to t_2}$$

$$(\_)\ \frac{\Gamma \vdash_n t_i \quad i \in m}{\Gamma \vdash_n (t_i | i \in m)} \qquad \langle\_\rangle\ \frac{\Gamma \vdash_{n+1} t}{\Gamma \vdash_n \langle t \rangle} \qquad \forall\ \frac{\Gamma, X^n \vdash_n t}{\Gamma \vdash_n \forall X.t} \qquad U\Rightarrow\ \frac{\Gamma \vdash_n t}{\Gamma \vdash_n U \Rightarrow t}$$

$$\textbf{var}\ \frac{\Gamma \vdash}{\Gamma \vdash_n x : t}\ x : t^n \in \Gamma \qquad \textbf{loc}\ \frac{\Gamma \vdash}{\Gamma \vdash_n l : L} \qquad \textbf{fun}\ \frac{\Gamma, x : t_1^n \vdash_n e : t_2}{\Gamma \vdash_n \lambda x : t_1.e : t_1 \to t_2}$$

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

$$\textbf{tuple}\ \frac{\Gamma \vdash \quad \{\Gamma \vdash_n e_i : t_i \mid i \in m\}}{\Gamma \vdash_n (e_i | i \in m) : (t_i | i \in m)} \qquad \textbf{proj}\ \frac{\Gamma \vdash_n e : (t_i | i \in m)}{\Gamma \vdash_n \pi_j\ e : t_j}\ j < m$$

$$\textbf{spawn}\ \frac{\Gamma \vdash_n e : () \to t}{\Gamma \vdash_n \mathit{spawn}\ e : ()} \qquad \textbf{new}\ \frac{\Gamma \vdash_n e : L \to t}{\Gamma \vdash_n \mathit{new}\ e : L} \qquad \textbf{input}\ \frac{\Gamma \vdash_n e : (L, U\Rightarrow t)}{\Gamma \vdash_n \mathit{input}\ e : t}$$

$$\textbf{output}\ \frac{\Gamma \vdash_n e : (L, t)}{\Gamma \vdash_n \mathit{output}\ e : ()} \qquad \textbf{run}\ \frac{\Gamma \vdash_n e : \langle t \rangle \quad \Gamma \vdash_n t}{\Gamma \vdash_n \mathit{run}\ e : t} \qquad \textbf{brck}\ \frac{\Gamma \vdash_{n+1} e : t}{\Gamma \vdash_n \langle e \rangle : \langle t \rangle}$$

$$\textbf{esc}\ \frac{\Gamma \vdash_n e : \langle t \rangle}{\Gamma \vdash_{n+1} \tilde{}e : t} \qquad \textbf{csp}\ \frac{\Gamma \vdash_n e : t}{\Gamma \vdash_{n+1} \%e : t} \qquad \textbf{poly}\ \frac{\Gamma, X^n \vdash_n e : t}{\Gamma \vdash_n \Lambda X.e : \forall X.t}$$

$$\textbf{spec}\ \frac{\Gamma \vdash_n e : \forall X.t_2 \quad \Gamma \vdash_n t_1}{\Gamma \vdash_n e\{t_1\} : t_2[X := t_1]}$$

$$\textbf{case}\ \frac{\{\Gamma \vdash_n \overline{e}(p_i) : L \mid i \in m\} \quad \{\Gamma, \Gamma^n(p_i) \vdash_n e_i : t \mid i \in m\}}{\Gamma \vdash_n (p_i \Rightarrow e_i | i \in m) : U \Rightarrow t}\ m > 0$$

Fig. 2. Type System

— $\Gamma \vdash$, i.e. Γ is a well-formed context
— $\Gamma \vdash_n t$, i.e. t is a well-formed type at level $n \geq 0$
— $\Gamma \vdash_n e : t$, i.e. e is a well-formed term of type t at level $n \geq 0$

Levels are typical of multi-level languages (like $\lambda^{\bigcirc}$ of (Dav96)). In a dynamically typed multi-stage language, like MetaKlaim (see also (SSP98)), type variables get bound at different stages of a computation, and thus well-formedness is level dependent not only for terms, but also for types. The declarations in a context Γ have the following meaning: X^n means that the type variable X ranges over types t at level n, while $x : t^n$ means that the term variable x ranges over *values* of type t at level n.

Figure 2 gives the typing rules. Most of them are the multi-level extension of standard typing rules. The only typing rules that deserve some comments are:

— The rule for type variables supports a form of cross-stage persistence (as in (SSP98)), namely an X declared at level m can be used at higher levels.
— The typing for *run* has the additional premise $\Gamma \vdash_n t$, since the other premise implies only that $\Gamma \vdash_{n+1} t$.
— Rule **(case)** uses some auxiliary notation, namely a context $\Gamma^n(p)$ and a sequence

$\overline{e}(p)$ of terms defined by induction on $p \in \mathsf{P}$ as follows:

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

4.1. *Basic Properties of the Type System*

Properties like Weakening and Substitution are typical of type systems. Substitution is particularly important, since it clarifies the meaning of different declarations, namely how the type and level assigned to a variable x constrain the terms that may safely replace it. Lemma 4.5 expresses a property of interest for multi-level languages, namely how the validity of a judgment is affected when (some of) the levels are incremented. The situation is more subtle when (some of) the levels are decremented (see Lemma 6.5).

Notation 4.1 (Notation and Convention used in the rest of the paper).

— $\Gamma \vdash J$ ranges over the possible judgments of the type system, thus J matches anything that can be on the right hand side of $\vdash$.

— Γ^{+1} is the context obtained from Γ by incrementing the level of each declaration by 1, i.e. X^n becomes X^{n+1} and $x : t^n$ becomes $x : t^{n+1}$. □

Lemma 4.2 (Weakening). The following rules are admissible

$$\frac{\Gamma_1, \Gamma_2 \vdash J}{\Gamma_1, X^m, \Gamma_2 \vdash J}\ X \text{ fresh} \qquad \frac{\Gamma_1 \vdash_m t \quad \Gamma_1, \Gamma_2 \vdash J}{\Gamma_1, x : t^m, \Gamma_2 \vdash J}\ x \text{ fresh}$$

Proof. By induction on the derivation of $\Gamma_1, \Gamma_2 \vdash J$. □

Lemma 4.3 (Substitution). The following rules are admissible

$$\frac{\Gamma_1 \vdash_m t \qquad \Gamma_1, X^m, \Gamma_2 \vdash J}{\Gamma_1, \Gamma_2[X := t] \vdash J[X := t]} \qquad \frac{\Gamma_1 \vdash_m e : t \qquad \Gamma_1, x : t^m, \Gamma_2 \vdash J}{\Gamma_1, \Gamma_2 \vdash J[x := e]}$$

Proof. By induction on the derivation of $\Gamma_1, X^m, \Gamma_2 \vdash J$ and $\Gamma_1, x : t^m, \Gamma_2 \vdash J$ respectively. □

The following lemma implies that the type of a closed term is necessarily closed. This property is quite subtle, for instance it fails if we extend the language with an exception $exn : t$, instead of a polymorphic constant $raise_{exn} : \forall X.X$.

Lemma 4.4 (Strengthening). The following rules are admissible

$$\frac{\Gamma_1, X^m, \Gamma_2 \vdash_n e : t}{\Gamma_1, \Gamma_2 \vdash_n e : t}\ X \notin \mathrm{FV}(\Gamma_2, e) \qquad \frac{\Gamma_1, x : t^m, \Gamma_2 \vdash_n e : t}{\Gamma_1, \Gamma_2 \vdash_n e : t}\ x \notin \mathrm{FV}(e)$$

Proof. By induction on the derivation of $\Gamma_1, X^m, \Gamma_2 \vdash J$ and $\Gamma_1, x : t^m, \Gamma_2 \vdash J$ respectively. □

Lemma 4.5 (Promotion). The following rules are admissible

$$\frac{\Gamma_1, \Gamma_2 \vdash}{\Gamma_1, \Gamma_2^{+1} \vdash} \qquad \frac{\Gamma_1, \Gamma_2 \vdash_n t}{\Gamma_1, \Gamma_2^{+1} \vdash_{n+1} t} \qquad \frac{\Gamma \vdash_n e : t}{\Gamma^{+1} \vdash_{n+1} e : t}$$

Proof. The first two rules are proved by mutual induction on the derivation of $\Gamma_1, \Gamma_2^{+1} \vdash$ and $\Gamma_1, \Gamma_2^{+1} \vdash_{n+1} t$. The third rule is by induction on the derivation of $\Gamma^{+1} \vdash_{n+1} e : t$. □

Remark 4.6. One can easily adapt a type inference algorithm for system F (e.g. see (Car97b)) to MetaKlaim. Namely, given Γ, e and n the algorithm either returns a t such that $\Gamma \vdash_n e : t$ is derivable (t is unique up to α-conversion), or fails when such a t does not exists. □

5. Operational Semantics

Following the Klaim computational paradigm, we define the operational semantics over nets.

Definition 5.1. A **net** $N \in \mathsf{Net} \triangleq \mu(\mathsf{L} \times (\mathsf{E}_0 + \mathsf{V}_0^0 + \{exn, \mathsf{err}\}))$ is a multi-set of pairs consisting of a locality l and either a process $p(e)$, or a value $u(v)$ in the tuple space, or exn indicating that a process at l has raised an exception, or err indicating that a process at l has crashed. □

The dynamics of a net is given by a relation $N \Longrightarrow N'$ defined in terms of two transition relations $e \overset{a}{\longmapsto} e'$ and $e \longmapsto exn \mid \mathsf{err}$ for terms[†]: err means that a process has crashed, this is different from node failure (that we do not model), and from a deadlocked process (e.g. a process that is waiting to input a tuple that never arrives); exn means that an exception has been raised (for simplicity we do not provide exception handling facilities, although in practice they are important). The transitions relations are defined in terms of evaluation contexts (see (WF94)) and reductions $r^0 \overset{a}{\longrightarrow} e'$ (and $r^0 \longrightarrow exn \mid \mathsf{err}$) for actions and $r^1 \overset{1}{\longrightarrow} e' \mid \mathsf{err}$ for symbolic evaluation.

Figure 3 summarizes the syntactic categories for the operational semantics. Redexes are the subterms where rewriting takes place. Evaluation contexts identify which of the redexes in a term should be evaluated first, namely the hole [] gives the position of such a redex. The syntax is complicated by the stratifications into levels (borrowed from MetaML).

5.1. *Reduction and Transition Relation*

Even if we are interested in defining $\longmapsto$ only on closed terms, we must consider open redexes because of evaluation under (dynamic) lambda. Figure 4 defines the reduction $\longrightarrow$ and uses the following auxiliary operations:

[†] We write $e \longmapsto exn \mid \mathsf{err}$ to denote $e \longmapsto exn$ or $e \longmapsto \mathsf{err}$ (and similarly for other transition relations).

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

$$\begin{array}{rcl} v^0 & ::= & l \mid \lambda x:t.e \mid (v_i^0 | i \in m) \mid \langle v^1 \rangle \mid \Lambda X.e \mid (vmr_i^0 | i \in m) \\ v^{n+1} & ::= & x \mid l \mid \lambda x:t.v^{n+1} \mid v_1^{n+1} v_2^{n+1} \mid \mathsf{fix}\ x:t.v^{n+1} \mid (v_i^{n+1} | i \in m) \mid \pi_j\ v^{n+1} \\ & \mid & op\ v^{n+1} \mid \langle v^{n+2} \rangle \mid \%v^n \mid \Lambda X.v^{n+1} \mid v^{n+1}\{t\} \mid (vmr_i^{n+1} | i \in m) \\ v^{n+2} & += & \tilde{}\, v^{n+1} \end{array}$$

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

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

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

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

$$\begin{array}{rcl} r^0 & ::= & x \mid v_1^0 v_2^0 \mid \mathsf{fix}\ x:t.e \mid \pi_j\ v^0 \mid op\ v^0 \mid \tilde{}\, e \mid \%e \mid v^0\{t\} \\ r^1 & ::= & \tilde{}\, v^0 \end{array}$$

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

$$\begin{array}{rcl} E_i^n & ::= & E_i^n e \mid v^n E_i^n \mid (\overline{v}^n, E_i^n, \overline{e}) \mid \pi_j\ E_i^n \mid op\ E_i^n \mid \langle E_i^{n+1} \rangle \mid E_i^n\{t\} \\ & \mid & (\overline{vmr}^n, Ep_i^n \Rightarrow e, \overline{mr}) \\ E_i^{n+1} & += & \lambda x:t.E_i^{n+1} \mid \mathsf{fix}\ x:t.E_i^{n+1} \mid \tilde{}\, E_i^n \mid \%E_i^n \mid \Lambda X.E_i^{n+1} \\ & \mid & (\overline{vmr}^{n+1}, vp^{n+1} \Rightarrow E_i^{n+1}, \overline{mr}) \\ E_i^i & += & [] \end{array}$$

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

— Actions $a \in \mathsf{A} ::= \tau \mid l:e \mid s(e) \mid i(v)@l \mid o(v)@l$ with $e \in \mathsf{E}_0$ and $v \in \mathsf{V}_0^0$

Fig. 3. Values, redexes and evaluation contexts

— Function $match(p, v^0)$ either returns a closed substitution $\rho : \mathsf{X} \xrightarrow{fin} \mathsf{V}_0^0$ or $fail$. Its definition is by induction on $p \in \mathsf{P}$. The base cases are:

p	$match(p, v^0)$	
$x!t$	$x := v^0$	if $\emptyset \vdash_0 v^0 : t$ and v^0 global, otherwise $fail$
$x = e$	$x := v^0$	if $v^0 \equiv e \in \mathsf{L}$, otherwise $fail$

match is used by *input* for dynamic type checking of *global* values (see Definition 3.4).

— *Demotion* $v^{n+1} \downarrow_n \in \mathsf{E}$ is defined by induction on $v^{n+1} \in \mathsf{V}^{n+1}$ (and $vp^{n+1} \in \mathsf{VP}^{n+1}$):

v^{n+1}	$v^{n+1} \downarrow_n \in \mathsf{E}$	
x	x	
$\langle v^{n+2} \rangle$	$\langle v^{n+2} \downarrow_{n+1} \rangle$	
$\tilde{}\, v^n$	$\tilde{}\, v^n \downarrow_{n-1}$	$(n > 0)$
$\%v^n$	$\%v^n \downarrow_{n-1}$	if $n > 0$
	$v^n[x := \%x \mid x \in \mathrm{FV}(v^n)]$	otherwise

vp^{n+1}	$vp^{n+1} \downarrow_n \in \mathsf{P}$
$x!t$	$x!t$
$x = v^{n+1}$	$x = v^{n+1} \downarrow_n$
vmr^{n+1}	$vmr^{n+1} \downarrow_n$
$vp^{n+1} \Rightarrow v^{n+1}$	$vp^{n+1} \downarrow_n \Rightarrow v^{n+1} \downarrow_n$

$(\lambda x : t.e)\ v_2^0 \xrightarrow{\tau} e[x := v_2^0]$
$v_1^0\ v_2^0 \longrightarrow \mathsf{err}$ if $v_1^0 \not\equiv \lambda x : t.e$

$\mathsf{fix}\ x : t.e \xrightarrow{\tau} e[x := \mathsf{fix}\ x : t.e]$

$\pi_j\ (v_i^0 | i \in m) \xrightarrow{\tau} v_j^0$ if $j < m$
$\pi_j\ v^0 \longrightarrow \mathsf{err}$ if $v^0 \not\equiv (v_i^0 | i \in m)$ with $j < m$

$spawn\ v^0 \xrightarrow{s(v^0())} ()$ if $v^0 \in \mathsf{V}_0^0$, otherwise exn (scope extrusion exception)

$new\ v^0 \xrightarrow{l:(v^0 l)} l$ if $v^0 \in \mathsf{V}_0^0$, otherwise exn (scope extrusion exception)

$output\ (l, v^0) \xrightarrow{o(v^0)@l} ()$ if $v^0 \in \mathsf{V}_0^0$, otherwise exn (scope extrusion exception)
$output\ v^0 \longrightarrow \mathsf{err}$ if $v^0 \not\equiv (l, v_1^0)$

$input\ (l, (vp_i^0 \Rightarrow e_i | i \in m)) \xrightarrow{i(v^0)@l} e_j[\rho]$ if $match(vp_j^0, v^0) = \rho$ for some $j \in m$
$input\ v^0 \longrightarrow \mathsf{err}$ if $v^0 \not\equiv (l, (vp_i^0 \Rightarrow e_i | i \in m))$

$run\ \langle v^1 \rangle \xrightarrow{\tau} v^1 \downarrow_0$ if $v^1 \in \mathsf{V}_0^1$, otherwise exn (demotion exception)
$run\ v^0 \longrightarrow \mathsf{err}$ if $v^0 \not\equiv \langle v^1 \rangle$

$(\Lambda X.e)\{t\} \xrightarrow{\tau} e[X := t]$
$v^0\{t\} \longrightarrow \mathsf{err}$ if $v^0 \not\equiv \Lambda X.e$

$x \longrightarrow \mathsf{err}$
$\tilde{}e \longrightarrow \mathsf{err}$
$\%e \longrightarrow \mathsf{err}$

$\tilde{}\langle v^1 \rangle \xrightarrow{1} v^1$
$\tilde{}v^0 \xrightarrow{1} \mathsf{err}$ if $v^0 \not\equiv \langle v^1 \rangle$

Fig. 4. Reductions for actions and symbolic evaluation

In all other cases $\_ \downarrow_n$ commutes with the top level term (and pattern) construct.

Remark 5.2. Intuitively, Demotion is like Compilation: it translates a value $\langle v^1 \rangle$ representing a program into an *executable* term $v^1 \downarrow_0$. However, the reduction for *run* performs demotion only when v^1 is closed, in order to prevent unresolved link errors (see Remark 3.3). □

We comment some of the reduction rules in Figure 4 (the others are standard):

— The rules for *spawn*, *new*, *output* and *input* come from Klaim, those for *run* and symbolic evaluation $\xrightarrow{1}$ come from MetaML.
— *spawn* (and similarly *new* and *output*) checks that the process spawned is closed (in order to prevent scope extrusion), and raises an exception otherwise.
— *input* is non-deterministic and requires pattern matching, which includes dynamic type-checking of global values. Moreover, *input* may get stuck, e.g. $input(l, x!X \Rightarrow e)$ is stuck because there are no closed values of type X.
— *run* checks that the value that is demoted is closed (in order to prevent unresolved links), and raises an exception otherwise.

— All reductions to err correspond to type- or level-errors. For instance, $\tilde{}e$ and $\%e$ are not well-typed at level 0, nor is x when all variables are declared at level > 0.

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

$$\frac{r^0 \xrightarrow{a} e'}{E_0^0[r^0] \xmapsto{a} E_0^0[e']} \qquad \frac{r^0 \longrightarrow exn \mid \mathsf{err}}{E_0^0[r^0] \longmapsto exn \mid \mathsf{err}} \qquad \frac{r^1 \xrightarrow{1} e'}{E_1^0[r^1] \xmapsto{\tau} E_1^0[e']} \qquad \frac{r^1 \xrightarrow{1} \mathsf{err}}{E_1^0[r^1] \longmapsto \mathsf{err}}$$

5.2. *Net Transition Relation*

The relation $\Longrightarrow$ is defined (in terms of $\longmapsto$) by the following rules

$$\frac{e \longmapsto exn}{N \uplus (l : p(e)) \Longrightarrow N \uplus (l : exn)} \qquad \frac{e \longmapsto \mathsf{err}}{N \uplus (l : p(e)) \Longrightarrow N \uplus (l : \mathsf{err})}$$

$$\frac{e \xmapsto{\tau} e'}{N \uplus (l : p(e)) \Longrightarrow N \uplus (l : p(e'))}$$

$$\frac{e \xmapsto{i(v^0)@l_2} e'}{N \uplus (l_1 : p(e)) \uplus (l_2 : u(v^0)) \Longrightarrow N \uplus (l_1 : p(e')) \uplus (l_2 : p(()))}$$

$$\frac{e \xmapsto{o(v^0)@l_2} e'}{N \uplus (l_1 : p(e)) \Longrightarrow N \uplus (l_1 : p(e')) \uplus (l_2 : u(v^0))}$$

$$\frac{e \xmapsto{s(e_2)} e_1}{N \uplus (l : p(e)) \Longrightarrow N \uplus (l : p(e_1)) \uplus (l : p(e_2))}$$

$$\frac{e \xmapsto{l_2 : e_2} e_1}{N \uplus (l_1 : p(e)) \Longrightarrow N \uplus (l_1 : p(e_1)) \uplus (l_2 : p(e_2))}\ l_2 \notin L(N) \cup \{l_1\}$$

where $L(N) \triangleq \{l \mid (l : \_) \in N\} \subseteq_{fin} \mathsf{L}$ is the set of localities in the net N. The rules have an obvious meaning, we just remark that the side condition in the last rule ensures freshness of l_2.

5.3. *Basic Properties of the Operational Semantics*

The following properties are straightforward to prove.

Lemma 5.3 (Reduction). If $r \xrightarrow{a} e'$ or $r \xrightarrow{1} e'$, then $\mathrm{FV}(e') \subseteq \mathrm{FV}(r)$.

Lemma 5.4 (Unique Decomposition). Given $n \in \mathsf{N}$ and $e \in \mathsf{E}$, then
— either $e \in \mathsf{V}^n$
— or exist (unique) $i \in \{0, 1\}$ and $E_i^n \in \mathsf{E}_i^n$ and $r^i \in \mathsf{R}^i$ such that $e \equiv E_i^n[r^i]$

Proof. By induction on the structure of $e \in \mathsf{E}$. □

Lemma 5.5 (Transition). If $e \in \mathsf{E}_0$ and $e \xmapsto{a} e'$, then $e' \in \mathsf{E}_0$.

Proof. Immediate from Lemmas 5.4 and 5.3. □

Lemma 5.6 (Net Transition). If $N \Longrightarrow N'$, then $L(N) \subseteq L(N')$.

6. Type Safety

In order to express the type safety results we introduce two notions of well-formed net: one is *global*, the other is relative to a subset L of nodes.

Definition 6.1 (Well-formed Net).

Global: A net N is well-formed $\stackrel{\Delta}{\Longleftrightarrow}$ $(l : \mathsf{err}) \notin N$, and for every $(l : p(e)) \in N$ exists t s.t. $\emptyset \vdash_0 e : t$.

Local: 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 $(l : p(e)) \in N$ with $l \in L$ exists t s.t. $\emptyset \vdash_0 e : t$.

□

In the definition of well-formed net nothing is said about values $u(v)$ in the tuple spaces, since they are considered untrusted. In fact, processes can fetch such values only through the *input* primitive, which performs dynamic type-checking.

Theorem 6.2 (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

The type safety theorem then guarantees that a well-formed net will never give rise to type- or level-errors. Together with dynamic type checking performed with input operations, these imply that our type system can be used for protecting hosts from imported code, thus ensuring various kinds of host security properties (as in (YH99b)).

Remark 6.3. The local type safety property is enforced by two features of MetaKlaim: the dynamic type-checking performed by the input operation (namely *match*), which prevents ill-typed values in tuple spaces to pollute well-typed processes; the absence of Klaim's *eval* primitive, which would allow processes external to L to spawn ill-typed processes at a locality in L. For instance, with an *eval* primitive similar to a 'remote' *spawn* the following net transition would become possible

$$l_{bad} : p(eval(l_{good}, v_{bad})), l_{good} : u(v) \Longrightarrow l_{bad} : p(()), l_{good} : p(v_{bad}()), l_{good} : u(v)$$

where v_{bad} is any closed value (at level 0) such that $v_{bad}() \longmapsto \mathsf{err}$. □

6.1. *Technical Lemmas for Type Safety*

The proof of Type Safety relies on the basic properties of the type system (see Section 4.1), and the following lemmas linking operational semantics and type system.

Notation 6.4 (Auxiliary definitions and notations used in this section).

— $\vdash a$ means that action a is well-formed. Action τ, $i(v)$ and $o(v)$ are always well-formed. Actions $s(e)$ and $l : e$ are well-formed, provided that $\emptyset \vdash_0 e : t$ for some t.
— $\Gamma^n(E_i^n)$ is the typing context for the hole in the evaluation context $E_i^n \in \mathsf{EC}_i^n$, and is defined by induction on $E_i^n \in \mathsf{EC}_i^n$ (and Ep_i^n)

E_i^n	$\Gamma^n(E_i^n) \in \mathsf{Ctx}$
$[]$	$\emptyset \quad (n = i)$
$\langle E_i^{n+1} \rangle$	$\Gamma^{n+1}(E_i^{n+1})$
$\overline{vmr}^n, Ep_i^n \Rightarrow e, \overline{mr}$	$\Gamma^n(Ep_i^n)$

Ep_i^n	$\Gamma^n(Ep_i^n) \in \mathsf{Ctx}$
$x = E_i^n$	$\Gamma^n(E_i^n)$

E_i^{n+1}	$\Gamma^{n+1}(E_i^{n+1}) \in \mathsf{Ctx}$
$\lambda x : t.E_i^{n+1}$	$x : t^{n+1}, \Gamma^{n+1}(E_i^{n+1})$
$\mathsf{fix}\ x : t.E_i^{n+1}$	$x : t^{n+1}, \Gamma^{n+1}(E_i^{n+1})$
$\Lambda X.E_i^{n+1}$	$X^{n+1}, \Gamma^{n+1}(E_i^{n+1})$
$\tilde{}E_i^n$	$\Gamma^n(E_i^n)$
$\%E_i^n$	$\Gamma^n(E_i^n)$
$\overline{vmr}^{n+1}, vp^{n+1} \Rightarrow E_i^{n+1}, \overline{mr}$	$\Gamma^{n+1}(vp^{n+1}), \Gamma^{n+1}(E_i^{n+1})$

In all other cases $\Gamma^n(\_)$ is applied to the immediate sub-context. $\Gamma^n(vp^n)$ is a special case of $\Gamma^n(p)$ defined in Section 4 (indeed, according to the grammars in Figures 1 and 3, VP^n is included in P).

— $e \not\longrightarrow \mathsf{err}$ means that $e \longrightarrow \mathsf{err}$ does not hold (and similarly $e \not\mapsto \mathsf{err}$).

Lemma 6.5 (Demotion). The following rules are admissible

$$\frac{\Gamma^{+1} \vdash}{\Gamma \vdash} \qquad \frac{\Gamma^{+1} \vdash_{n+1} t}{\Gamma \vdash_n t} \qquad \frac{\Gamma^{+1} \vdash_{n+1} v^{n+1} : t}{\Gamma \vdash_n v^{n+1} \downarrow_n : t}$$

Proof. The first two rules are proved by mutual induction on the derivation of $\Gamma^{+1} \vdash$ and $\Gamma^{+1} \vdash_{n+1} t$. The third rule is by induction on the derivation of $\Gamma^{+1} \vdash_{n+1} v^{n+1} : t$. □

Lemma 6.6 (Structure). If $\Gamma \vdash_0 v^0 : t$, then one of the following possibilities holds:

— $v^0 \equiv l$ and $t \equiv L$
— $v^0 \equiv (v_i^0 | i \in m)$ and $t \equiv (t_i | i \in m)$ with $\Gamma \vdash_0 v_i^0 : t_i$ for all $i \in m$
— $v^0 \equiv \lambda x : t_1.e$ and $t \equiv t_1 \to t_2$ with $\Gamma, x : t_1^0 \vdash_0 e : t_2$
— $v^0 \equiv \langle v^1 \rangle$ and $t \equiv \langle t' \rangle$ with $\Gamma \vdash_1 v^1 : t'$
— $v^0 \equiv \Lambda X.e$ and $t \equiv \forall X.t'$ with $\Gamma, X^0 \vdash_0 e : t'$
— $v^0 \equiv (vp_i^0 \Rightarrow e_i | i \in m)$ and $t \equiv U \Rightarrow t'$ with $\Gamma, \Gamma^0(vp_i^0) \vdash_0 e_i : t'$ for all $i \in m$

Proof. By case analysis on the last rule in the derivation of $\Gamma \vdash_0 v^0 : t$. Because of the structure of $v^0 \in \mathsf{V}^0$ we have to consider only the following cases (see Figure 2): (loc), (fun), (tuple), (brck), (poly) and (case). □

Lemma 6.7 (Match). If $\rho = match(p, v^0)$, then $\emptyset \vdash_0 \rho(x) : t$ and $\rho(x)$ is global when $x : t^0 \in \Gamma^0(p)$.

Proof. By induction on the structure of $p \in \mathsf{P}$.

— Base case $x!t$. ρ is $x := v^0$ and $\Gamma^0(p)$ is $x : t$. The property follows immediately from the definition of *match*.
— Base case $x = e$. ρ is $x := v^0 \in \mathsf{L}$ and $\Gamma^0(p)$ is $x : L$. The property follows immediately from $v^0 \in \mathsf{L}$.
— Inductive step $(p_i | i \in m)$. $\Gamma^0(p) \equiv \Gamma^0(p_0), \ldots, \Gamma^0(p_{m-1})$ and v^0 must be of the form $(v_i^0 | i \in m)$.
$\rho = \rho_0, \ldots, \rho_{m-1}$ with $\rho_i = match(p_i, v_i^0)$. If $x : t^0 \in \Gamma^0(p_j)$, then $\rho(x) = \rho_j(x)$ and the property follows by the induction hypothesis for p_j.

□

Lemma 6.8 (Safety and Subject Reduction for $\longrightarrow$).

— If $\Gamma^{+1} \vdash_0 r^0 : t$, then $r^0 \not\longrightarrow \mathsf{err}$ and $r^0 \xrightarrow{a} e'$ implies $\Gamma^{+1} \vdash_0 e' : t$ and $\vdash a$.
— If $\Gamma^{+1} \vdash_1 r^1 : t$, then $r^1 \not\longrightarrow {}^1\mathsf{err}$ and $r^1 \xrightarrow{1} e'$ implies $\Gamma^{+1} \vdash_1 e' : t$.

Proof. By induction on the derivation of $\Gamma^{+1} \vdash_i r^i : t$. The last rule in the derivation uniquely determines (the structure of) r^i.

— (var) contradicts that r^i is a redex, because all x declared in Γ^{+1} are at a level > 0
— (loc), (fun), (tuple), (brck), (csp), (poly) and (case) contradict that r^i is a redex
— (app) implies $r^0 \equiv v_1^0 v_2^0$ and $\Gamma^{+1} \vdash_0 v_1^0 : t_1 \to t$ and $\Gamma^{+1} \vdash_0 v_2^0 : t_1$. By Lemma 6.6 v_1^0 must be of the form $\lambda x : t_1.e$ and $\Gamma^{+1}, x : t_1^0 \vdash_0 e : t$, therefore $r^0 \xrightarrow{\tau} e[x := v_2^0]$. By Lemma 4.3 we get $\Gamma^{+1} \vdash_0 e[x := v_2^0] : t$.
— (fix) implies $r^0 \equiv \mathsf{fix}\ x : t.e \xrightarrow{\tau} e[x := \mathsf{fix}\ x : t.e]$ and $\Gamma^{+1}, x : t^0 \vdash_0 e : t$. By Lemma 4.3 (and $\Gamma^{+1} \vdash_0 \mathsf{fix}\ x : t.e : t$) we get $\Gamma^{+1} \vdash_0 e[x := \mathsf{fix}\ x : t.e] : t$.
— (proj) implies $r^0 \equiv \pi_j\ v^0$ and $\Gamma^{+1} \vdash_0 v^0 : (t_i | i \in m)$ with $j < m$ and $t \equiv t_j$. By Lemma 6.6 v^0 must be of the form $(v_i^0 | i \in m)$ with $\Gamma^{+1} \vdash_0 v_i^0 : t_i$ for $i \in m$, therefore $r^0 \xrightarrow{\tau} v_j$.
— (spawn) implies $r^0 \equiv spawn\ v^0$, $t \equiv ()$, and $\Gamma^{+1} \vdash_0 v^0 : () \to t'$. If $v^0 \in \mathsf{V}_0^0$, then $r_0 \xrightarrow{s(v^0())} ()$ and $\emptyset \vdash_0 v^0 : () \to t'$. By (app) we get $\emptyset \vdash_0 v^0() : t'$, i.e. $\vdash a$. Otherwise *exn*.
— (new) is similar to (spawn).
— (output) implies $r^0 \equiv output\ v^0$, $t \equiv ()$ and $\Gamma^{+1} \vdash_0 v^0 : (L, t')$. By Lemma 6.6 v^0 must be of the form (l, v_0^0) with $\Gamma^{+1} \vdash_0 v_0^0 : t'$. If $v_0^0 \in \mathsf{V}_0^0$, then $r^0 \xrightarrow{o(v_0^0)@l} ()$. Otherwise *exn*.
— (input) implies $r^0 \equiv input\ v^0$ and $\Gamma^{+1} \vdash_0 v^0 : (L, U \Rightarrow t)$. By Lemma 6.6 v^0 must be of the form $(l, (vp_i^0 \Rightarrow e_i | i \in m))$ with $\Gamma^{+1}, \Gamma^0(vp_i^0) \vdash_0 e_i : t$ for all $i \in m$. Therefore, the possible reductions are $r^0 \xrightarrow{i(v_0^0)@l} e_j[\rho]$ with $match(vp_j^0, v_0^0)$ for some $j \in m$. By Lemma 6.7 and repeated application of Lemma 4.3 we get $\Gamma^{+1} \vdash_0 e_j[\rho] : t$.
— (run) implies $r^0 \equiv run\ v^0$, $\Gamma^{+1} \vdash_0 v^0 : \langle t \rangle$ (and $\Gamma^{+1} \vdash_0 t$). By Lemma 6.6 v^0 must be of the form $\langle v^1 \rangle$ with $\Gamma^{+1} \vdash_1 v^1 : t$. If $v^1 \in \mathsf{V}_0^1$, then $r^0 \xrightarrow{\tau} v^1 \downarrow_0$ and $\emptyset \vdash_1 v^1 : t$ by Lemma 4.4. Therefore, by Lemma 6.5 we get $\emptyset \vdash_0 v^1 \downarrow_0 : t$. Otherwise *exn*.

— (spec) implies $r^0 \equiv v^0\{t_1\}$ and $\Gamma^{+1} \vdash_0 v^0 : \forall X.t_2$ and $\Gamma^{+1} \vdash_0 t_1$ with $t \equiv t_2[X := t_1]$. By Lemma 6.6 v^0 must be of the form $\Lambda X.e$ with $\Gamma^{+1}, X^0 \vdash_0 e : t_2$. Thus $r^0 \xrightarrow{\tau} e[X := t_1]$, and by Lemma 4.3 we get $\Gamma^{+1} \vdash_0 e[X := t_1] : t$.
— (esc) implies $r^1 \equiv \tilde{} v^0$ and $\Gamma^{+1} \vdash_0 v^0 : \langle t \rangle$. By Lemma 6.6 v^0 must be of the form $\langle v^1 \rangle$ with $\Gamma^{+1} \vdash_1 v^1 : t$, and thus $r^0 \xrightarrow{\tau} v^1$.

□

Lemma 6.9 (Replacement for Evaluation Context). If $E_i^n \in \mathsf{EC}_i^n$ and $\Gamma \vdash_n E_i^n[e] : t$, then exists $t' \in \mathsf{T}$ such that
— $\Gamma, \Gamma^n(E_i^n) \vdash_i e : t'$
— $\Gamma, \Gamma^n(E_i^n) \vdash_i e' : t'$ implies $\Gamma \vdash_n E_i^n[e'] : t$.

Proof. By induction on the structure of $E_i^n \in \mathsf{EC}_i^n$ and the derivation of $\Gamma \vdash_n E_i^n[e] : t$. □

Lemma 6.10 (Safety and Subject Reduction for $\longmapsto$). If $\emptyset \vdash_0 e : t$, then $e \not\longmapsto \mathsf{err}$ and $e \xrightarrow{a} e'$ implies $\emptyset \vdash_0 e' : t$ and $\vdash a$.

Proof. By Lemma 5.4 either $e \in \mathsf{V}^0$ (and there is nothing to prove) or $e \equiv E_i^0[r^i]$ for some i. In the latter case, by Lemma 6.9, we have $\Gamma^0(E_i^0) \vdash_i r^i : t'$ for some t'. Thus we can apply Lemma 6.8, since $\Gamma^n(E_i^n)$ is always of the form Γ^{+1} (see Notation 6.4). □

Finally we prove Theorem 6.2.

Proof. We give the details only for the proof of Local Type Safety (the proof of Global Type Safety is similar). The proof is by case-analysis on the rule used to derive $N \Longrightarrow N'$. The only interesting cases correspond to net transitions that involve at least one locality in L:
— if $e \longmapsto exn$, then the safety property remains trivially true
— if $e \longmapsto \mathsf{err}$, then $l \notin L$ because of Lemma 6.10
— if $e \xrightarrow{\tau} e'$ and $l \in L$, then e' is well-typed at level 0 by Lemma 6.10
— if $e \xrightarrow{i(v^0)@l_2} e'$ and $l_1 \in L$, then e' is well-typed at level 0 by Lemma 6.10
— if $e \xrightarrow{o(v^0)@l_2} e'$ and $l_1 \in L$, then e' is well-typed at level 0 by Lemma 6.10, and whether $l_2 \in L$ is irrelevant for the safety property
— if $e \xrightarrow{s(e_2)} e_1$ and $l \in L$, then e_1 and e_2 are well-typed at level 0 by Lemma 6.10
— $e \xrightarrow{l_2:e_2} e_1$ and $l \in L$, then e_1 is well-typed at level 0 by Lemma 6.10, also e_2 is well-typed, but it is irrelevant for local safety because $l_2 \notin L$.

□

7. Examples

In this section, we exemplify the use of MetaKlaim to program WAN applications. Each example is presented in a simplified form, but addresses a significant aspect in WAN programming. The first example, *group communication*, deals with generation of

lightweight efficient components implementing a form of broadcast remote communication. The second example, *nomadic data collector*, addresses the issue of protecting host machines from mobile code that travels along the net for retrieving information on a piece of data. The third example, *dynamic linker and loader*, illustrates separation of concerns supported by generative components. In the rest of this section, we will freely use ML-like notations for functions, local declarations, datatypes, lists, conditional and sequential composition. Moreover, for type-setting reasons, we write `fn x:t.e` instead of $\lambda x : t.e$ and `V X.t` instead of $\forall X.t$.

7.1. *Group Communication*

We introduce a function `grout` that implements a form of *group communication*: a message (the parameter of the function) is broadcasted to each locality of a given list statically known. Function `grout` is a simple example of multipoint applications (e.g. audio/video applications) which exploit multicast communications. In fact, function `grout` can be thought of as a basic building block for constructing more sophisticated applications which permit, e.g., to dynamically change the group of receivers or to hierarchically structure the group (like in distributed mailing lists). We make use of the following types:

```
L                                       (* localities *)
type Data = ...                         (* type of messages *)
type GO = L List -> Data -> ()          (* type of group output *)
type GOs = L List -> <Data> -> <()>     (* type for staged group output *)
type GOcg = L List -> <Data -> ()>      (* type for group output code generator *)
```

We first present a version of `grout` that does not use staging. As it is expected, function `grout` takes a list of localities `l` and a message `x` as arguments and outputs a tuple containing `x` at each locality in `l`. Notice that to be well-typed the message has to be a global value.

```
fun grout (l:L List, x:Data):() =
        if l=nil then ()
        else output(hd l,x) ; grout(tl l,x)
```

This version of `grout` does not take advantage of the fact that its parameters are available at different stages of the computation. Indeed, the fact that the list parameter `l` is statically available (while the message parameter `x` will be available at run time) offers an opportunity to optimize the code of the function with respect to `l`. In this way, the overhead of looking up the first element of `l`, and of recursively calling the function on the tail of `l` each time a message has to be sent, can be removed. Following a general *staging* method (see (CMS02; TS00; She01)), we define a staged version `grout_s` of `grout`

```
fun grout_s (l:L List, x:<Data>):<()> =
        if l=nil then <()>
        else <output(%(hd l),~x) ; ~(grout_s(tl l,x))>
```

The types reflect the fact that the list parameter is available in the first stage, and the message is available in the second stage. The brackets around the branches of the `if`-expression means that the function `grout_s` returns code. In the else branch, the `output`

operation is delayed to the second stage, while the recursive call to `grout_s` is performed in the first stage. The staging annotations in `output(%(hd l),~x)` have been inserted because the value of `hd l` is computed in the first stage but used later, while `x` has type code. The staged version of `grout` is used to define a function `grout_cg` that takes a list `l` of localities and generates specialized code for broadcasting a message to all localities in `l`.

```
(* code generator *)
fun grout_cg (l:L List):<Data -> ()> = <fn x:Data . ~(grout_s l <x>)>
(* grout specialized for a list l *)
fun grout_l:Data -> () = run(grout_cg l)
```

For instance, when `grout_cg` is applied to a list of localities `[l1,l2]`, we get

```
< fn x:Data . output(l1,x) ; output(l2,x) ; () >
```

If we have an application that reads messages from `orig` and broadcast them to the same list `dest` of localities, then one could use the broadcast specialized for `dest`

```
fix p:(). input(orig, x!Data => grout(dest,x) ; p)
fix p_o:(). input(orig, x!Data => grout_dest(x) ; p_o)     (* optimized p *)
```

The main advantage of `p_o` over `p` is better performance.

Adaptive applications can benefit from code generation, as described in (HS01). The following application extends `p` with a new functionality, that permits to change the list of destinations

```
fun ap (dest:L List):() =
    input(orig, x!Data   => grout(dest,x) ; (ap dest)
              | l!L List => (ap l))
```

One can obtain the process `ap dest` from a general template `ap_gen` parameterized w.r.t. a function `do` specifying what to do when an input `x:X` is received, and a function `upd` for updating what to do when an input `y:Y` is received

```
fun ap_gen (do:X->(), upd:Y->X->()):() =
    input(orig, x!X => (do x) ; ap_gen(do,upd)
              | y!Y => ap_gen(upd y,upd))
```

More precisely `ap dest` amounts to `ap_gen(upd dest, upd)` where `upd` is such that `upd y x = grout(y,x)`. However, we can exploit the code generator `grout_cg` for defining a different updating function `upd y = run(grout_cg y)`, that returns a *better* `do`.

7.2. *Nomadic Data Collector*

We now address the issue of protecting host machines from possibly malicious mobile code. 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 *)
type Key  = ...      (* authorization keys *)
type Data = ...
(* polymorphic types of local operations input, output, spawn *)
type I = V X. (L,U=>X) -> X
type O = V X. (L,X) -> ()
type S = V X. (() -> X) -> ()
(* polymorphic types of meta-operations for input, output, spawn *)
type MI = Key -> <I>
type MO = Key -> <O>
type MS = <S>
(* code abstractions with static security checks *)
type MEnvK = (L,MI,MO,MS)
type CAK = MEnvK -> <()>
```

The 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) is 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 location 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` (that is checked in the example below by a function `safe`), the meta-operation `in' k` could generate an actual `input` without run-time overhead, or `()` when the key does not allow to read anything (or customized run-time checks, that we do not detail). Customization of the other local operations can be done similarly.

```
fun in' (k:Key):<I> = if safe k then <input> else <()>
```

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 where all 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`. In the definition of `pca(k,i,u)` cross-stage persistence is used to hard-wire the parameters `i` and `u` at the appropriate level.

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

```
      ~(in' k) {()}
          (%self', (_=%i, l!L) =>
                   ~(out' k) {CAK} (l,%ca)) ; q>
```

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 a customized environment `env`, and finally activates the resulting code.

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

7.3. *Dynamic Linking and Loading*

We now present the MetaKlaim implementation of a basic facility to dynamically load components. We have already pointed out that a key issue of most WAN applications is the ability to control the loading policy of components. One may want either to load components just-when-needed, or prefer to fetch in advance all components requested by a certain application. The (naive) solution is to parameterize applications with respect to a linker, and call the linker whenever a component (or service) is needed. However, this does not ensure enough flexibility. A better approach is to define a generative component parameterized with respect to a meta-linker. The meta-linker can decide whether to load a requested component at code-generation-time, by immediately invoking the linker, or to postpone the loading at run-time, namely by generating code for a call to the linker.

Hereafter, we view a linker as a specialized component which, given a name of a service, either succeeds in establishing a connection between the service and the calling application by returning an authorization key, or raises an exception. Indeed, we are not interested in the details of the linker, but only in its abstract behavior. We make use of the following basic types:

```
type Key = ...      (* authorization keys *)
type Name = ...     (* service names *)
```

A linker behaves like a function: given the name of the component returns the authorization key required to exploit component functionalities. Hence, a linker has the type

```
type Linker = Name -> Key   (* linkers *)
```

A *meta-linker* is an higher order component which given the name of a component returns a *frozen* authorization key: the metadata information which has to be actualized to provide the linking step. Hence, the meta-linker delays the computation which performs the linking of the component. The meta-linker has the type

```
type MLinker = Name -> <Key>   (* metalinkers *)
```

Finally, a parameterized component is a component whose linking policy has not been fixed in advance. To behave properly a meta-linker must be supplied to the parameterized component. Hence, the type of a parameterized component is:

```
type PC = MLinker -> <()>     (* parameterized components *)
```

A directory (i.e. repository) of components in MetaKlaim takes the form of a tuple space. Parameterized components are distinguished tuples stored in the tuple space: they are tuples having type `PC`.

The `input` operation is the basic facility to find a component inside a directory. It queries the tuple space to find the required component: *pattern-matching* is used to select components according to their types . For instance, the expression

```
input (self, x!PC => spawn(() => e)
```

queries the local tuple space to select a parameterized component of type `PC`.

The following process `execute` fetches a parameterized component from the local tuple place, generates code by supplying it a meta-linker, and then spawns a process that executes the generated code.

```
fun Mexecute (self:L, mlinker:MLinker):() =
  fix exec:().
      input (self, x!PC=> spawn(() => run(x mlinker)) ; exec)
```

An invocation of the meta-linker will be of the form `<...~(mlinker n)...>`. The escape operator `~e` is used to insert delayed computation into larger computation. Using the meta programming facilities, the programmer has a fine grain control on the evaluation order of the program. For instance, the meta-linker can decide whether to invoke the linker at code generation time, i.e. `mlinker n = <%(linker n)>`, or whether to generate code for invoking the linker at run-time, i.e. `mlinker n = <%linker %n>`. In the first case, `linker n` is statically evaluated; if the linker fails to make a connection, the code of the application will not be executed at all.

Notice that the `MExecute` cannot be transmitted over the net. It is not a mobile component since it will not get through the dynamic checking of the pattern matching. The critical point is that the `input` operation could have been programmed to download a malicious component from a remote, untrusted, host.

A mobile `Meta-Execute` can be programmed as follows

```
fun MMExecute (self:L, mlinker:MLinker, in', spawn', run'):() =
  fix exec:().
      in' (self, x!PC=> spawn'(() => run'(x mlinker)) ; exec)
```

The type of `MMExecute` is parameterized with respect to the locality (where the code will be executed) and the local operations exploiting the full power of system F.

8. Related Work

There are several approaches related with some of the issues tackled by MetaKlaim. Some of these approaches have been mentioned in the first two sections of the paper, here we only consider the most strictly related one and draw comparisons with some process languages equipped with process distribution and higher-order remote communication.

Kali Scheme (CJK95) is a distributed implementation of Scheme (SS75) which adds to facilities for program distribution and for communication of higher order objects, such

as procedures and closures. These features, and the *continuation passing* programming style inherited from Scheme, fit the language also for applications dealing with dynamic code transmission, linking and generation. However, differently from MetaKlaim, the language does not have a type system and type errors are detected at run-time.

$D\pi\lambda$ (YH99a) is the result of integrating the call-by-value λ-calculus, the π-calculus, and primitives for process distribution and remote process creation. Differently from MetaKlaim, communication is synchronous and channel based, and localities are anonymous. This last feature implies that localities cannot be explicitly referred by processes and do not have a first-class status, thus the distributed fragment of the calculus is not as expressive as MetaKlaim. $D\pi\lambda$ permits the transmission of process abstractions parameterized with respect to resource (i.e. channel) names. The language is equipped with a type system that, by constraining values that may be output, statically guarantees that processes willing to perform inputs at a given channel are co-located. The type system for $D\pi\lambda$ is completely static and relies on restrictions laid on values that are sent across localities, whilst the type system for MetaKlaim uses dynamic checks for controlling the type of values that are received by input actions. Moreover, for a $D\pi\lambda$ system to type check all of its subsystems have to type check (this burden is partially alleviated from the fact that subsystems can be type checked independently and then composed while checking compatibility of their use of resources), while a MetaKlaim net can also be partially checked relatively to a subset of localities (in the same spirit as (RH99)). These features indicate that the MetaKlaim approach better fits also open and, possibly, untrusted large scale distributed systems where it is important to protect hosts from imported code.

(YH99b) extends to $D\pi\lambda$ the type system of (YH02) for an higher order variant of the π-calculus. This type system permits controlling the effect of transmitted process abstractions on local resources (i.e. channels). All the remarks about the first type system also apply to this one. Furthermore, differently from MetaKlaim, processes are assigned fine-grain types that, like interfaces, record the resources to which processes have access together with the corresponding capabilities, and process abstractions are assigned dependent functional types that abstract from channel names and types. Although process abstractions are not polymorphic as in MetaKlaim, channel names may appear and be bound both in terms and in types and thus, in some sense, play the role of MetaKlaim type variables.

Confined-λ (Kir01) is an higher-order functional language that supports distributed computing by allowing expressions at different localities to communicate via channels. In Confined-λ, authors of code can assign regions (i.e. subsystems) to values in order to limit the part of a system where a value can freely move. Then, a type system is defined that statically guarantees that each value can roam only within the corresponding region. Differently from our approach, communication is channel based, localities cannot be dynamically created, the transmissible process abstractions can be parameterized with respect to channel names, and the type of a transmissible value restricts the subsystem where the value can freely move. The Confined-λ type system is completely static and

relies on restrictions laid on values whenever they are used as arguments of output operations, while the MetaKlaim type system relies on dynamic checks whenever input operations are performed. Moreover, differently from MetaKlaim, a Confined-λ program may execute only if all the expressions it contains are well-typed. In conclusion, while the MetaKlaim approach better fit open and untrusted large scale distributed systems, the Confined-λ approach is more suitable for guaranteing secrecy properties, i.e. that a given (secret) information is not leaked outside a fixed subsystem, in small scale distributed systems.

Additional examples of type systems for distributed higher order functional languages can be found in (Kir02). In addition to the previous considerations, we can also say that they all use static effect systems for approximating dynamic properties, while MetaKlaim uses global values, which is a naive way of saying that processes have no effect annotations in their type.

9. Conclusions and Future Work

We have described MetaKlaim, a foundational calculus for global computing. Our approach is based on polymorphic types á la system F, staging constructs (á la MetaML), and dynamic type checking. To the best of our knowledge, MetaKlaim is the first calculus that integrates multi-stage features with primitives for mobile distributed processes. In this paper we have mainly focussed on developing the foundations of its mathematical theory, but we believe that our programming notations can form the core of a real programming language with facilities for code mobility and multi-stage programming. The consistency of operational semantics and type system implies that in our approach hosts are protected from imported code, thus ensuring various kinds of host security. This, together with the possibility of partially checking MetaKlaim nets only relatively to subsets of localities, make MetaKlaim suitable for programming applications in open and, possibly, untrusted large scale distributed systems.

Process calculi with distribution and mobility other than Klaim could in principle be enriched with staging and meta-programming constructs. However, if some properties must be guaranteed, mobile code should be dynamically checkable and, possibly, customizable. This in practice requires the exploitation of higher-order remote communication (to split code migration into code exchange and code spawning) and code types at the object level, and does not seem to fit well with process calculi with just name passing.

There are several directions one can consider for improving MetaKlaim, for instance

— We could replace Klaim's flat structure of localities with hierarchical (dynamically changing) tree structure typical of Ambients (CG00b). Each ambient will have a tuple space, a pool of local processes, and a pool of (mobile) sub-ambients.
— We could refine the type system into a type-and-effect system (TJ94), and extend it with dynamics (ACPR95; Dug99).
— We could introduce guardians for monitoring node activities (FMP02), in addition to the dynamic checks performed by processes (during an *input*-action).

As a future work we plan to implement MetaKlaim by possibly exploiting the Klaim prototype implementation as a starting point. The Klaim prototype implementation can be downloaded from the Klaim homepage (KHP98), while its detailed description, together with some programming examples, can be found in (BDP02). Moreover, we also plan to develop realistic WAN applications to gather more data for further validating the advantages of the approach and to assess our linguistic choices (for example, different forms of output could be envisaged that differ for the amount of performed checks and, then, for the secrecy properties guaranteed).

Acknowledgements. We thank the anonymous referees for their useful comments and criticisms, and Didier Remy for his advise to look at dynamics in statically typed languages.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Mobile Code. *Software — Practice and Experience*, 32:1365–1394, 2002.

L. Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, 15–17 January 1997.

L. Cardelli. Type systems. In A.B.Jr Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.

L. Cardelli. 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, pages 51–94. Springer-Verlag, 1999.

N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4):444–458, 1989.

L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science special issue on Coordination*, 240(1), July 2000.

L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, number 1378 of Lecture Notes in Computer Science, pages 140-155, Springer, 1998.

H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.

C. Calcagno, E. Moggi, and T. Sheard. Closed types for safe imperative MetaML. *To appear in Journal of Functional Programming*, 2002.

Microsoft Corporation. *Microsoft DCOM Technical Overview*, 2001. Available on-line at `http://www.microsoft.com/ntserver/techresources/appserv/COM/dcomtec.asp`.

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

R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science special issue on Coordination*, 240(1):215–254, 2000.

S. Drossopoulou. Towards an abstract model of java dynamic linking and verification. In *Proc. of The Third ACM SIGPLAN Workshop on Types in Compilation*, Montreal, Canada, September 21, 2000. ACM.

D. Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, 1999.

U.W. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

C. Fournet, 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)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

G. Ferrari, E. Moggi, and R. Pugliese. Guardians for ambient-based monitoring. In V. Sassone, editor, *F-WAN: Foundations of Wide Area Network Computing*, number 66 in ENTCS. Elsevier Science, 2002.

A. Fuggetta, G. Picco, and G. Vigna. Understanging code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

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

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de doctorat d'etat, University of Paris VII, 1972.

M. Hennessy and J. Riely. Distributed processes and location failures. *Theoretical Computer Science*, 266(1–2):693–735, September 2001.

B. Harrison and T. Sheard. Dynamically adaptable software with metacomputations in a staged language. In W. Taha, editor, *Proc. of the Int. Work. on Semantics, Applications, and Implementations of Program Generation (SAIG)*, volume 2196 of *LNCS*, pages 163–182. Springer-Verlag, 2001.

M. Hicks and S. Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, 2000.

M. Hicks, 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*, volume 2071 of *LNCS*, pages 147–176. Springer-Verlag, 2000.

S. Kamin, M. Callahan, and L. Clausen. Lightweight and generative components II: Binary-level components. In *(Tah00)*, pages 28–50, 2000.

The Klaim Home Page, 1998. Provides source code, applications and documentation online at `http://music.dsi.unifi.it/klaim.html`.

Z.D. Kirli. Confined mobile functions. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 283–294, Washington - Brussels - Tokyo, June 2001. IEEE Computer Society.

Z.D. Kirli. *Mobile Computation with Functions*. Kluwer, April 2002.

A. Kennedy and D. Syme. The design and implementation of generics for the .net common language runtime. In *Proc. ACM SIGPLAN Conference on Programming Language Desing and Implementation (PLDI)*, Snowbird, Utah, USA, june 2001. ACM Press.

G. Morrisett and N. Glew. Type-safe linking and modular assembly language. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261. ACM, 1999.

The MetaML Home Page, 2000. Provides source code and documentation online at `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.

SUN Microsystems. *Java Beans API Specification*, 2002. Available on-line at `http://java.sun.com/products/javabeans/docs/spec.html`.

D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Professional Computing Series. Addison Wesley, 1996.

J.C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation, Paris*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, NY, june 1974. Springer-Verlag. Extension of typed lambda calculus to user-defined types and polymorphic functions.

J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104. ACM, 1999. Full version as CogSci Report 4/98, University of Sussex, Brighton.

G.-C. Roman, G.P. Picco, and A.L. Murphy. Software engineering for mobility. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 241–260, NY, June 4–11 2000. ACM Press.

F.B. Schneider. Enforceable security policies. *ACM Transation on Information and System Security*, 3(1):30–50, 2000. Also appeared as Technical Report TR99-1759, Department of Computer Science, Cornell University, Jul 1999.

P. Sewell. Modules, abstract types, and distributed versioning. In C. Norris and Jr.J.B. Fenwick, editors, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 236–247, New York, January 17–19 2001. ACM Press.

T. Sheard. Using MetaML: A staged programming language. In *Lecture Notes in Computer Science*, volume 1608, pages 207–239, 1999.

T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Proc. of the Int. Work. on Semantics, Applications, and Implementations of Program Generation (SAIG)*, volume 2196 of *LNCS*, pages 2–46. Springer-Verlag, 2001.

F.B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back. Conference on the Occasion of Dagstuhl's 10th Anniversary*, number 2000 in Lecture Notes in Computer Science, pages 86–101. Springer-Verlag, 2000.

G.J. Sussman and G.L. Steele. Scheme: An Interpreter for Extended Lambda Calculus. Technical Report AI Memo, no. 349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.

M. Shields, T. Sheard, and S.L. Peyton Jones. Dynamic typing as staged type inference. In ACM, editor, *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, New York, NY, USA, 1998. ACM Press. San Diego, California, 19–21 January 1998.

P. Sewell and P.T. Wojciechowski. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April/June 2000.

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

J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of Symposium on Partial Evaluation and Semantics Based Program manipulation*, pages 203–217. ACM SIGPLAN, 1997.

W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Proc. Workshop on Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 47–77. Springer-Verlag, 1999.

A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In J.C.M. Baeten and S. Mauw, editors, *Proc. of the Int. Conf. on Concurrency Theory (CONCUR)*, volume 1664 of *LNCS*, pages 557–572. Springer-Verlag, 1999.

N. Yoshida and M. Hennessy. Assigning types to processes. CogSci Report 99.02, School of Cognitive and Computing Sciences, University of Sussex, UK, 1999.

N. Yoshida and M. Hennessy. Assigning types to processes. *Information and Computation*, 174(2):143–179, 2002.