Mobile Functions and Secure Information Flow

Dilsun Kırlı

Laboratory for Foundations of Computer Science, The University of Edinburgh,
King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, UK.
Fax : +44 131 667 72 09, Email : zdk@dcs.ed.ac.uk

Abstract. We introduce a higher-order mobile code language called Mobile-λ which combines a sequential fragment inherited from the functional language ML with a distributed computation model based on communication via synchronous channels. We explore the question of secrecy in Mobile-λ by means of a type and effect system. On the one hand, we contribute to the field of type-based approaches to security by bringing into attention a computational model which has not yet been studied in this context. On the other hand, we show that the type and effect discipline continues to be of interest to languages with a functional core. Our type system exploits the ideas behind this discipline to enforce secrecy while also allowing types to be parametric in secrecy levels.

Keywords: ML, Facile, mobile functions, information flow, type and effect discipline.

1 Introduction

Programming languages which facilitate mobility of code are becoming widely used since the mobile agents paradigm appears to be well-suited for computation in the highly distributed networks of our day. Agents are defined as code containing computational objects which can move from site to site and can be executed on the sites they visit using the local resources. These resources may be of different sensitivity levels and if no precaution is taken, high sensitivity data faces the risk of being leaked or corrupted. In the security literature [7, 2, 19] leaking is often referred to as a violation of *secrecy* and corruption as a violation of *integrity*. In order to protect against such violations, it is essential to constrain the behaviour of agents based on a set of criteria which indicate potentiality for harmful behaviour. It is important to note that this problem does not only concern controlling access to local resources but also tracing the flow of information through computation.

Several authors point out the benefits of addressing security concerns such as secrecy and integrity in the design phase of a mobile code language as opposed to delegating them entirely to the lower-levels of the system. The latter brings about ad hoc approaches in providing security as discussed in [18, 3, 6]. Recently, the approach of formulating secure flow analyses as type systems has earned attention [19, 16, 13, 5, 11, 9]. In cases where the language is intended for computation in closed systems, a property formulated as a type soundness definition can be enforced through strong static typing for all well-formed programs of the language. Languages which support computation in open systems, however, may have to employ different strategies which may involve static typing, dynamic typing or combinations of these.

In this paper, we focus on the problem of providing secrecy within the framework of a higher-order mobile code language called Mobile-λ which has its roots in languages such as Concurrent ML [12] and Facile [4]. Mobile-λ supports mobile computation by combining a functional core with constructs for transmitting and receiving values on channels across remote sites. A characteristic feature is that functions and channels are first class values and mobile agents can simply be coded as mobile functions. The type-based approach to secure information flow has so far addressed several computational models. Among these are the models provided by various imperative languages studied in [19, 16, 9] and functional languages such as the Secure Lambda Calculus (SLam) [5], an extension of the λ-calculus for trust analysis [11] and the Dependency Core Calculus [1]. Some authors have also extended their work to address concurrency. A concurrent version of SLam which uses a shared memory for communication and the imperative language studied in [13] are examples. The languages which are based on the λ-calculus are closely related to ours. However, none of them adopts message passing communication such as ours. Moreover, the relation of Mobile-λ to computation with mobile agents is more evident than any of these languages.

In the rest of the paper, we introduce the language Mobile-λ, discuss an appropriate notion of secrecy for it and motivate a property which we wish to enforce. We focus on information flow aspects rather than access control. Following the type-based approach to security we then propose a type system to enforce this property. Types are enriched with effect annotations which capture potential behaviour of functions. Our selection of annotations allows us to detect potentially harmful flows of information by examining the type inferred for each expression of the language. Our type system is a direct descendant of the former type and effect systems proposed for ML [8, 14] and Facile [15] which dealt with the problem of providing polymorphism in the presence of features for imperative and concurrent computation and those which were used for communication analyses [10]. A significant advantage of using the type and effect discipline is that providing ML-style polymorphism in this new context, namely allowing the expression of code which behaves uniformly over values of different secrecy levels, becomes a natural extension of the earlier work in this field.

2 Mobile-λ

In this section we introduce Mobile-λ briefly commenting on the characteristic features of its abstract syntax and dynamic semantics.

2.1 Syntax

Expressions which serve as constructors are labeled with secrecy levels: H for secret and L for public. Special constants and primitive operators are ranged over by *scon* and *op* respectively. Channels are parameterised over types.

$$
\begin{array}{lll}
\text{(label)} & l ::= H \mid L \\
\text{(type)} & t ::= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{chan}(t) \mid t \rightarrow t \\
\text{(expression)} & e ::= scon^l \mid x \mid \texttt{chan}^l(t) \mid \texttt{fn}^l\, x \Rightarrow e \mid \texttt{rec}^l\, f(x) \Rightarrow e \\
& \quad \mid e_1\,\texttt{op}\,e_2 \mid e_1\, e_2 \mid e_1; e_2 \mid \texttt{let}\, x = e_1\,\texttt{in}\,e_2 \\
& \quad \mid \texttt{if}\, e_1\,\texttt{then}\,e_2\,\texttt{else}\,e_3 \mid e_1? \mid e_1!e_2
\end{array}
$$

2.2 Dynamic Semantics

Values of the language consist of primitive values, channel identifiers and function closures, represented by the semantic objects $val(scon^l)$, ci and $\langle l, E, x, e\rangle$ respectively. We assume that the dynamic representation of these objects encodes the secrecy label of their constructing expression. A channel identifier ci is represented by a pair (l, n) where n can be considered as a freshly generated number at each invocation of a channel allocation expression. An evaluation environment E is defined as a finite map from identifiers to values. A judgement of the form $E \vdash_s e \stackrel{w}{\Longrightarrow} val$ states that expression e evaluates to value val against an environment E at site s. From here on in the paper, we omit the site annotation from the turnstile when there is no scope for confusion.

The evaluation relation $\Longrightarrow$, which is defined in Figure 1, is annotated with the possible communication events which may occur during evaluation; $\mathsf{new}\; ci$, $ci\,?\,val$, $ci\,!\,val$ representing the allocation of a channel, receiving and sending values over a channel respectively. Note that the communication event of a purely sequential computation which does not involve any communication is represented by ϵ. Initially, E contains the ubiquitous values and other values that are provided as predefined at the site of evaluation. The evaluation rules are quite intuitive. Note that the rule **(chan)** creates a new channel identifier which is globally unique. APPLY in rule **(op)** serves as a semantic tool to define the application of a predefined operation to an argument.

Mobile-λ does not include a construct for parallel composition of expressions. The concurrent activity is revealed in the composition of evaluations at each site. The top-level evaluation decomposes into two threads of evaluation in which expression e_1 is evaluated at site s_1 and expression e_2 is evaluated at site s_2.

$$
\frac{E_1 \vdash_{s_1} e_1 \stackrel{w_1}{\Longrightarrow} val_1 \quad E_2 \vdash_{s_2} e_2 \stackrel{w_2}{\Longrightarrow} val_2 \quad w_1 \mid\mid w_2 \hookrightarrow w \quad IOmatch(w_1, w_2)}{(E_1, E_2), (s_1[e_1] \mid\mid s_2[e_2]) \stackrel{w}{\Longrightarrow} (val_1, val_2)}
$$

Mobile-λ adopts synchronous communication. The synchronisation between the expressions is specified by a collection of matching rules between communication event sequences given in Figure 2. The predicate $IOmatch$ defined below expresses the fact that the concurrent evaluation must respect these matching rules.

(scon) $$E \vdash scon^l \stackrel{\epsilon}{\Longrightarrow} val(scon^l)$$

(chan) $$E \vdash \mathtt{chan}^l(t) \stackrel{\mathsf{new}\,ci}{\Longrightarrow} ci \quad ci = (l, n), n \,\text{new}$$

(var) $$E \vdash x \stackrel{\epsilon}{\Longrightarrow} E(x)$$

(fn) $$E \vdash \mathtt{fn}^l\, x \;\Rightarrow e \stackrel{\epsilon}{\Longrightarrow} \langle l, E, x, e\rangle$$

(rec) $$\frac{val = \langle l, E[f \;\mapsto\; val],\, x,\, e\rangle}{E \vdash \mathtt{rec}^l\, f(x) \Rightarrow e \stackrel{\epsilon}{\Longrightarrow} val}$$

(op) $$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} val_1 \quad E \vdash \mathtt{op} \stackrel{\epsilon}{\Longrightarrow} b \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val_2}{E \vdash e_1\, \mathtt{op}\, e_2 \stackrel{w_1.w_2}{\Longrightarrow} \mathrm{APPLY}(b, val_1, val_2)}$$

(app) $$\frac{\begin{array}{l} E \vdash e_1 \stackrel{w_1}{\Longrightarrow} \langle l', E',\, x,\, e\rangle \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val \\ E'[x \;\mapsto\; val] \vdash e \stackrel{w_3}{\Longrightarrow} val' \end{array}}{E \vdash e_1\, e_2 \stackrel{w_1.w_2.w_3}{\Longrightarrow} val'}$$

(seq) $$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} val_1 \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val_2}{E \vdash e_1\, ;\, e_2 \stackrel{w_1.w_2}{\Longrightarrow} val_2}$$

(let) $$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} val_1 \quad E[x \;\mapsto\; val_1] \vdash e_2 \stackrel{w_2}{\Longrightarrow} val_2}{E \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \stackrel{w_1.w_2}{\Longrightarrow} val_2}$$

(if-t) $$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} true^l \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val}{E \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \stackrel{w_1.w_2}{\Longrightarrow} val}$$

(if-f) $$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} false^l \quad E \vdash e_3 \stackrel{w_2}{\Longrightarrow} val}{E \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \stackrel{w_1.w_2}{\Longrightarrow} val}$$

(receive) $$\frac{E \vdash e \stackrel{w}{\Longrightarrow} ci}{E \vdash e? \stackrel{w.ci?val}{\Longrightarrow} val}$$

(send) $$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} ci \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val \quad ci = (l, n)}{E \vdash e_1!e_2 \stackrel{w_1.w_2.ci!val}{\Longrightarrow} val(unit^l)}$$

Fig. 1. Sequential Evaluation Rules

Definition 1 (IOmatch). *$IOmatch(w_1, w_2)$ if and only if $w_1 \mid\mid w_2 \hookrightarrow w$ and all annotations in w are of the form* $\mathsf{new}\, ci$ *for some ci.*

$$\epsilon \mid\mid \epsilon \hookrightarrow \epsilon$$

$$\frac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1.com \mid\mid w_2 \hookrightarrow w.com} \qquad \frac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1 \mid\mid w_2.com \hookrightarrow w.com}$$

$$\frac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1.ci!val \mid\mid w_2.ci?val \hookrightarrow w} \qquad \frac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1.ci?val \mid\mid w_2.ci!val \hookrightarrow w}$$

Fig. 2. Event Matching

3 Information Flow in Mobile-λ

Noninterference: It is a common approach to define security requirements of a system based on an interface model which specifies restrictions on a system's input-output relation. A frequently sought secrecy property is *noninterference* which requires that high-level user inputs cannot interfere with low-level outputs. In other words, the values of public outputs do not depend on the values of secret inputs. The formulation of this property depends strongly on the characteristics of the system in question and what is assumed to be observable by whom. These issues have been discussed in detail by several authors [7, 17, 18]. Various notions of noninterference for deterministic systems and their possibilistic and probabilistic versions for nondeterministic systems have been proposed. These notions may also be of relative strength, depending on the assumptions about whether termination or nontermination of a program can be observed. For illuminating examples the reader is referred to [5, 18].

The type system which we present in the following section enforces a noninterference property for mobile functions. This property is formulated by considering a single thread of evaluation at a specific site. Ultimately, we would like to establish a robust and general notion for noninterference for mobile computation with functions. This would involve including in Mobile-λ a language feature to spawn expressions and consider in depth issues arising from concurrency. This paper constitutes the first step in this direction, exploring the applicability of the type and effect discipline in tracing information flow in a higher-order functional language with channel based communication.

Closer Look at Mobile-λ: We noted that all values in Mobile-λ are first class, in other words they are treated uniformly. In accordance with this principle, all values are classified into two levels of secrecy and tracing the flow of information involves tracing the flow of functions, channels and basic values. Functions are central to computation in Mobile-λ and it is essential for a secrecy property defined in terms of an interface model to specify a relation between the values flowing into and out of a function. In Mobile-λ, a datum of information can flow into a function through its environment and argument. As well as accessing other functions in the environment through bindings of its free identifiers, a function may interact with its environment by communicating on channels. Therefore, the appropriate notion of noninterference would require that the following do not depend on the secret values which flow into a function

- a public result returned by the function;
- the set of the public values which flow out of the function on communication channels.

Channels are different from basic values in that they are instrumental in the flow of values within the system. It is natural to expect their classification to have some relevance to the secrecy levels of the values they are allowed to transmit. Although Mobile-λ does not include this as a run-time check in its dynamic semantics, a reasonable static semantics would determine allowable forms of communication on a channel. We restrict our attention to the case where channels transmit the values of the same secrecy levels as themselves.

A challenging point in the enforcement of the noninterference property arises from the presence of conditional statements. This is an issue widely studied in the literature. There is always an implicit flow from the guard of a conditional expression to its branches. The first example below shows how a secrecy violation may occur if the public result of a conditional expression depends on the value of a secret guard. This problem can be solved by imposing these kind of conditional expressions to return secret values only. In Mobile-λ, however, this would not be sufficient. As we see in the second example, the value of the guard which is to be kept secret might be leaked through a communication on a public channel such as ch. This point should be taken into consideration by our type system.

1. $\texttt{if}\ true^H\ \texttt{then}\ true^L\ \texttt{else}\ false^L$
2. $\texttt{if}\ true^H\ \texttt{then}\ ((ch\ !\ true^L)\ ;\ true^H)\ \texttt{else}\ ((ch\ !\ false^L)\ ;\ false^H)$

4 Type System

The type system presented in this section aims at enforcing the noninterference property discussed in the previous section. To simplify the presentation we restrict the polymorphism to secrecy labels only.

(Secrecy label)	l	$::=$	$H \mid L \mid \gamma$
(Raw type)	$\bar{\tau}$	$::=$	$\mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{chan}[\bar{\tau}] \mid \tau \xrightarrow{\kappa} \tau$
(Annotated type)	τ	$::=$	$(\bar{\tau}, l)$
(Effect)	κ	$::=$	$\emptyset \mid \mathsf{new}(l, \bar{\tau}) \mid \mathsf{send}(l, \bar{\tau}) \mid \mathsf{recv}(l, \bar{\tau}) \mid \kappa \cup \kappa$
(Type scheme)	σ	$::=$	$\forall \gamma_1 ... \gamma_n . \tau$

Our type system views types τ as a pair consisting of two components: a raw type $\bar{\tau}$ and a secrecy label l. Raw types classify values in the the conventional sense whereas secrecy labels associate them with a particular secrecy level. The secrecy labels can be considered as elements of a lattice with least element L and greatest element H where the ordering is denoted by the symbol $\sqsubseteq$ such that $L \sqsubseteq H$. We also use communication effects κ as annotations on function types. These stand for the communication events which may be incurred by a function when it is applied. The effect $\mathsf{new}(l, \bar{\tau})$ represents the effect of allocating a new channel of secrecy level l which carries values of the raw type $\bar{\tau}$, similarly the effects $\mathsf{send}(l, \bar{\tau}), \mathsf{recv}(l, \bar{\tau})$ represent sending and receiving values of the designated type $\bar{\tau}$ on a channel of secrecy level l. The effects can be merged by set union.

4.1 Example

The type of the function f in the below example indicates that, given a channel ch and an integer i, f sends i over ch. The polymorphism in the label allows f to be used for transmitting a secret integer at one point in the program and a public one at another.

$$\begin{array}{l} \mathtt{let}\, f = \mathtt{fn}^L\ ch \Rightarrow \mathtt{fn}\ i \Rightarrow ch\ !\ i \\ \mathtt{in}\, f\ (\mathtt{chan}^H(\mathtt{int}))\ 5^H; f\ (\mathtt{chan}^L(\mathtt{int}))\ 3^L \end{array}$$

$f : \forall \gamma.(\mathsf{chan}[\mathsf{Int}], \gamma) \xrightarrow{\emptyset} (\mathsf{Int}, \gamma) \xrightarrow{\kappa} (\mathsf{Unit}, \gamma)$ where $\kappa = \mathsf{send}(\gamma, \mathsf{Int})$.

4.2 Type Generalisation

The operations defined below are familiar from polymorphic type systems for languages with a functional core. The generalisation is allowed only on those label variables which are not free in the type environment or the effect associated with the expression.

Definition 2 ($\succ$). *A type scheme σ generalises a type τ, written as $\sigma \succ \tau$, if τ can be obtained from the body of σ by substituting secrecy labels l for the bound label variables γ. A substitution θ denotes a map from label variables to labels.*

Definition 3 (Generalisation). *Types are generalised to type schemes by the operation Gen.*
$Gen(\kappa, \Gamma)(\tau) = \mathtt{let}\{\gamma_1 ... \gamma_n\} = fv(\tau) \setminus (fv(\kappa) \cup fv(\Gamma))\, \mathtt{in}\, \forall \gamma_1 ... \gamma_n . \tau$
where fv computes the free variables in the expected way.

4.3 Typing Rules

The context in which an expression is associated with a type and an effect is represented by a static environment Γ, which maps value identifiers to type schemes σ. The static semantics for the language assigns a type and an effect (τ, κ) to each expression. This is represented by a judgement of the form $\Gamma \vdash e : \tau, \kappa$. The typing rules are given in Figure 3.

The typing rules **(fn)** and **(rec)** for function abstractions constrain the secrecy level of the result of a function to be at least as secret as that of the function itself. Otherwise, a public value could be obtained by the application of a secret function present in the environment. This would conflict with our restraint that public values should not depend on secret values. The same reasoning applies to the communication effects of the function. We allow functions to send on channels which are at least as secret as themselves. To formalise the latter condition we define the predicate called *Safe* below.

Definition 4 (Safe Flow). *$Safe(\kappa, l)$ if and only if for all $\mathsf{send}(l', \bar{\tau}) \in \kappa$ it is the case that $l \sqsubseteq l'$.*

Note that we make use of this predicate also in the typing rule **(if)** for the conditional expression. In that rule we use it to enforce that branches of an expression can only send on channels which are at least as secret as the guard of that expression. We have motivated this by an example at the beginning of this section. As is the case for ML-style typing, the generalisation of types to type schemes is performed by the **(let)** rule.

4.4 Nontermination

In Section 3 we mentioned that leakage of secret information may occur in rather subtle ways exploiting nontermination of programs. The following has been inspired by an example in [5]. Suppose that the following Mobile-λ expressions are evaluated in parallel. The value communicated on the channel *publicBool* would depend on the secret boolean guarding the conditional expression.

e1: $\texttt{let}\ loop1 = \texttt{rec}^H\ loop1\ () \Rightarrow \texttt{if}\ someBool\ \texttt{then}\ ()^H\ \texttt{else}\ loop1\ ()$
$\quad \texttt{in}\ loop1\ ()\ ;\ (publicBool\ !\ true^L)$

e2: $\texttt{let}\ loop2 = \texttt{rec}^H\ loop2\ () \Rightarrow \texttt{if}\ someBool\ \texttt{then}\ ()^H\ \texttt{else}\ loop2\ ()$
$\quad \texttt{in}\ loop2\ ()\ ;\ (publicBool\ !\ false^L)$

We argue that we can address problems of this kind by modifying the typing rule for recursive expressions. We could restrict the conditional expressions occurring in a recursive expression to branch on public values only. At this stage however, we have not incorporated this into the type system as we are also considering alternative approaches.

5 Formal Properties of the Type System

5.1 Consistency

We first prove the consistency of the static and dynamic semantics. We define a consistency judgement of the form $w : \kappa, CE \models val : (\bar{\tau}, l)$ which relates objects of the dynamic semantics to objects of the static semantics with respect to a given channel environment CE defined as a map from channel identifiers to secrecy labels and raw type pairs: $CE ::= [ci_1 \mapsto (l_1, \bar{\tau}_1), \ldots, ci_n \mapsto (l_n, \bar{\tau}_n)]$.

(scon)	$\Gamma \vdash scon^l : (typeof(scon), l), \emptyset$
(chan)	$\Gamma \vdash \mathtt{chan}^l(t) : (\mathsf{chan}[\bar{\tau}], l), \mathsf{new}(l, \bar{\tau})$
(var)	$\dfrac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau, \emptyset}$
(fn)	$\dfrac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \quad \tau_2 = (\bar{\tau}_2, l'), \kappa \quad l \sqsubseteq l' \quad Safe(\kappa, l)}{\vdash \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa} \tau_2, l), \emptyset}$
(rec)	$\dfrac{\Gamma[f \mapsto (\tau_1 \xrightarrow{\kappa} \tau_2)][x \mapsto \tau_1] \vdash e : \tau_2, \kappa \quad \tau_2 = (\bar{\tau}_2, l') \quad l \sqsubseteq l' \quad Safe(\kappa, l)}{\Gamma \vdash \mathtt{rec}^l\, f\,(x) \Rightarrow e : (\tau_1 \xrightarrow{\kappa} \tau_2, l), \emptyset}$
(app)	$\dfrac{\Gamma \vdash e_1 : (\tau_1 \xrightarrow{\kappa} \tau_2, l'), \kappa' \quad \Gamma \vdash e_2 : \tau_1, \kappa''}{\Gamma \vdash e_1\, e_2 : \tau_2, \kappa \cup \kappa' \cup \kappa''}$
(op)	$\dfrac{\Gamma \vdash e_1 : (\bar{\tau}, l), \kappa \quad \Gamma \vdash \quad e_2 : (\bar{\tau}, l'), \kappa'}{\Gamma \vdash e_1\, \mathsf{op}\, e_2 : (\bar{\tau}', l \sqcup l'), \kappa \cup \kappa'}$
(let)	$\dfrac{\Gamma \vdash e_1 : \tau, \kappa \quad \Gamma[x \mapsto Gen(\kappa, \Gamma)(\tau)] \vdash e_2 : \tau_2, \kappa'}{\Gamma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \tau_2, \kappa \cup \kappa'}$
(seq)	$\dfrac{\Gamma \vdash e_1 : \tau_1, \kappa \quad \Gamma \vdash e_2 : \tau_2, \kappa'}{\Gamma \vdash e_1\, ;\, e_2 : \tau_2, \kappa \cup \kappa'}$
(if)	$\dfrac{\begin{array}{l}\Gamma \vdash e_1 : (\mathsf{Bool}, l), \kappa, \quad \Gamma \vdash e_2 : (\bar{\tau}, l'), \kappa' \quad \Gamma \vdash e_3 : (\bar{\tau}, l'), \kappa'' \\ l \sqsubseteq l' \quad Safe(\kappa, l) \quad Safe(\kappa', l)\end{array}}{\Gamma \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : (\bar{\tau}, l'), \kappa \cup \kappa' \cup \kappa''}$
(receive)	$\dfrac{\Gamma \vdash e : (\mathsf{chan}[\bar{\tau}], l), \kappa}{\Gamma \vdash e? : (\bar{\tau}, l), \kappa \cup \mathsf{recv}(l, \bar{\tau})}$
(send)	$\dfrac{\Gamma \vdash e_1 : (\mathsf{chan}[\bar{\tau}], l), \kappa \quad \Gamma \vdash e_2 : (\bar{\tau}, l), \kappa'}{\Gamma \vdash e_1!e_2 : (\mathsf{Unit}, l), \kappa \cup \kappa' \cup \mathsf{send}(l, \bar{\tau})}$
(subs)	$\dfrac{\Gamma \vdash e : \tau, \kappa \quad \kappa \subseteq \kappa'}{\Gamma \vdash e : \tau, \kappa'}$

Fig. 3. Typing Rules

We adapt the proof method of [14] to our setting. The typing consistency relation that we seek to define must be viewed as the maximal fixed point of the property defined below. We consider an appropriate functional F and by verifying that F is monotonic conclude that the definition below is satisfied by its maximal fixed point. This part of the proof is omitted from our discussion.

Definition 5 **($w : \kappa, CE \models val : (\bar{\tau}, l)$).**

$w : \kappa, CE \models val(scon^l) : (typeof(scon), l)$

$w : \kappa, CE \models ci : (\mathsf{chan}[\bar{\tau}], l)$ *where* $ci = (l, n)$

if $(l, \bar{\tau}) \in Rng(\kappa), CE(ci) = (l, \bar{\tau})$

and for any val *such that* $ci\,?\,val \in w$ *or* $ci\,!\,val \in w$

$w : \kappa, CE \models val : (\bar{\tau}, l)$

$w : \kappa, CE \models \langle l, E, x, e \rangle : (\tau_1 \xrightarrow{\kappa'} \tau_2, l)$

if there exists a Γ *such that* $\Gamma \vdash \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa'} \tau_2, l), \emptyset$,

and $w : \kappa, CE \models E : \Gamma$

We write $w : \kappa, CE \models val : \forall \gamma_1 ... \gamma_n . \tau$ *if and only if* $w : \kappa, CE \models val : \theta\tau$ *for any substitution* θ *defined on the universally quantified variables of* τ.

Definition 6 **($w : \kappa, CE \models E : \Gamma$).**

$w : \kappa, CE \models E : \Gamma$

iff $Dom(E) = Dom(\Gamma)$,

and $w : \kappa, CE \models E(x) : \Gamma(x)$ *for any* $x \in Dom(E)$

Definition 7 (Extension). *(κ', CE') extends (κ, CE), which is denoted by $(\kappa, CE) \sqsubseteq (\kappa', CE')$ if and only if $\kappa \subseteq \kappa', Dom(CE) \subseteq Dom(CE')$ and for all $ci \in Dom(CE)$, if $CE(ci) \in Rng(\kappa)$ or $CE'(ci) \in Rng(\kappa)$ then $CE'(ci) = CE(ci)$.*

Definition 8 (Succession). *$(w, \kappa, CE) \sqsubseteq (w', \kappa', CE')$ if and only if $w' = w.w''$ for some w'', $(\kappa, CE) \sqsubseteq (\kappa', CE')$, and $w : \kappa, CE \models val : \tau$ implies $w' : \kappa', CE' \models val : \tau$ for any val and τ.*

Theorem 1 (Consistency). *We suppose that the system is composed of two expressions e_1 and e_2 which are evaluated at sites s_1 and s_2 respectively and are well-typed according to the given typing rules. We also assume that the initial evaluation environment at each site is consistent with the initial static environment where the initial state of the system is represented by w_0 and the initial channel environment is CE.*

$E_1 \vdash_{s_1} e_1 \overset{w_1}{\Longrightarrow} val_1,\ E_2 \vdash_{s_2} e_2 \overset{w_2}{\Longrightarrow} val_2,\ IOmatch(w_1, w_2),$

$\Gamma_1 \vdash_{s_1} e_1 : \tau_1, \kappa_1,\ \Gamma_2 \vdash_{s_2} e_2 : \tau_2, \kappa_2,$

$w_0 : \kappa_0, CE \models E_1 : \Gamma_1$ *and* $w_0 : \kappa_0, CE \models E_2 : \Gamma_2$

then there exists a CE' such that

1. $(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1, \kappa_0 \cup \kappa_1, CE')$ *and* $w_0.w_1 : \kappa_0 \cup \kappa_1, CE' \models val_1 : \tau_1$

2. $(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_2, \kappa_0 \cup \kappa_2, CE')$ *and* $w_0.w_2 : \kappa_0 \cup \kappa_2, CE' \models val_2 : \tau_2$

Proof (Consistency). The proof is given by induction on the inference of the matching $w_1 \mid\mid w_2 \hookrightarrow w$. □

5.2 Information Flow

Theorem 2 formalises our notion of noninterference. If we evaluate a well-typed expression at a given site against two different initial environments which are the same except at bindings for secret values and both evaluations terminate yielding a public value, these values are guaranteed to be equal. Moreover, the two evaluations agree on the values transmitted on public channels.

Theorem 2 (Noninterference). *Suppose that the following hold for a typing environment Γ, evaluation environments E and E', an initial state w_0 and a channel environment CE.*

$$
\begin{array}{l}
Dom(\Gamma) = Dom(E) = Dom(E') \\
E(x) = E'(x) \text{ for all } x \text{ such that } \Gamma(x) = L. \\
w_0 : \kappa_0, CE \models E : \Gamma,\ w_0 : \kappa_0, CE \models E' : \Gamma \\
E \vdash e \overset{w}{\Longrightarrow} val,\ E' \vdash e \overset{w'}{\Longrightarrow} val' \\
\Gamma \vdash e : (\bar{\tau}, l), \kappa
\end{array}
$$

then

1. *if $l = L$ then $val = val'$*
2. *$S \subseteq S'$ or $S \subseteq S'$ where $S = \{val \mid ci\,!\,val \in w, ci = (L, n)\}$ and $S' = \{val \mid ci\,!\,val \in w', ci = (L, n)\}$.*

Proof (Noninterference). The proof is given by induction on the depth of dynamic evaluation. □

6 Discussion and Conclusions

The literature on security is vast, including extensive studies of various aspects of security spanning a long period of time since the early days of computing. The wide acceptance of the mobile agents paradigm has introduced new challenges to the field and a number of new approaches have emerged to tackle these issues. In our work, we focus on a particular model for mobile computation which builds upon functional computation. We identify the challenges posed by the characteristics of this model in terms of providing secrecy. We then demonstrate that expressive type systems with rigorous underpinnings, which have been traditionally associated with functional computation, can be extended to address these challenges. The language we consider is rather skeletal at present. Yet, it is general enough to investigate the issues we have taken as subject to this work. Having established the basis presented in this paper, we are now considering stronger notions of noninterference and the possibility of exploiting the type and effect based analysis further in this context.

Acknowledgements I am grateful to Stephen Gilmore for his helpful comments on this work. My research is funded by a University of Edinburgh studentship.

References

1. M. Abadi, A. Banarjee, N. Heintze, and J.G. Riecke. A core calculus of dependency. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 147–160. ACM Press, 1999.
2. D. E. Denning. A lattice model of secure infromation flow. *Communications of the ACM*, 19(5):236–242, 1976.
3. J. Feigenbaum and P. Lee. Trust management and proof-carrying code in secure mobile-code applications. Presented at DARPA Workshop on Foundations of Secure Mobile Code, March 1997.
4. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
5. N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, CA, USA, 1998.
6. X. Leroy and F. Rouaix. Security properties of typed applets. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 391–400. ACM Press, 1998.
7. J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.
8. R. Milner, M. Tofte, R.Harper, and D.MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.
9. A. C. Myers and B. Liskov. Practical mostly-static information flow control. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, 1999.
10. H. R. Nielson and F. Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer-Verlag, 1997.
11. J. Palsberg and P. Ørbæk. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.
12. J. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.
13. G. Smith and D. Volpano. Secure infomation flow in a multi-threaded imperative language. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, San Diego, CA, USA, 1998.
14. J.P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.
15. B. Thomsen. Polymorphic sorts and types for concurrent functional programs. In J. Glauert, editor, *Proceedings of the 6th International Workshop on the Implementatin of Functional Languages*, Norwich,UK, 1994.
16. D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT '97*, pages 607–621, Lille, France, 1997.
17. D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, 1998.
18. D. Volpano and G. Smith. Language issues in mobile program security. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes of Computer Science*, pages 25–43. Springer Verlag, 1998.
19. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow anaylsis. *Journal of Computer Security*, 4(3):1–21, 1996.