Confined Mobile Functions

Z. Dilsun Kırlı
Laboratory for Foundations of Computer Science
Division of Informatics, The University of Edinburgh
King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK
Email: zdk@dcs.ed.ac.uk

Abstract

We introduce a higher-order language called Confined-λ which combines a functional sequential core with primitives for distributed computation. Confined-λ adopts a communication model based on synchronous message passing along channels. We focus on the problem of giving programmers control over the flow of values created by their programs. We show that by using a type-based approach a value can be forced to remain within the boundaries of a particular subsystem – a mobility region – specified by the programmer. The static type system presented in this paper guarantees that the values of well-typed Confined-λ programs will be confined to their specified mobility regions at run-time.

1 Introduction

Distributed computing systems are typically built to facilitate the sharing of hardware, software and information resources provided at separate sites in a network. Many modern systems bring together computing devices of different processing powers, software provided by different sources and information with different secrecy and integrity requirements. Moreover, the users which interact with the system may be of different trust levels. An important concern for systems such as these is to achieve effective exploitation of resources by distributing the computation among different sites without conflicting with the security goals set for the system.

A programming language which enables programmers to constrain the distribution of computation and the flow of information throughout the system would support the development of applications which exhibit controlled behaviour in terms of security and performance. In this paper, we consider a simple programming language – Confined-λ – which allows programmers to declare a *mobility region* for the services and the information they provide. The mobility region of an entity determines the subsystem in which it can flow freely. We propose a static type-system for our language which enforces the property called *confinement in a mobility region*. This property guarantees that the entities created and manipulated by well-typed Confined-λ programs will remain within their specified mobility regions. Throughout this paper, we will be concerned with confinement in a mobility region from the perspective of secrecy. By using Confined-λ users can create a piece of secret information and rely on the fact that this secret information will not be leaked to those parties which they consider ineligible to obtain it.

The computational model of Confined-λ is based on that of Facile [11] and Concurrent ML [20]. It has a core set of features inherited from the call-by-value functional language ML [16] and it supports channel-based communication between remote sites in a distributed system. All values in Confined-λ including functions and channels are first-class; they can be passed as arguments to functions, returned as results or communicated over channels. The fact that functions are code-containing objects makes Confined-λ a mobile code language. A function can be sent over a channel and at the receiving end it can be applied to a locally provided argument to obtain a result. In an application based on the design paradigm of code on demand [10], services would be implemented as functions and the requesters would access these functions by retrieving them over a channel.

The literature on programming language theory provides a base of concepts and techniques which can be exploited in formulating and proving secrecy properties of programs [1]. The recent years have seen an active interest in exploiting traditional program analysis techniques to address the security of information flow within computing systems. For example, control and data flow analyses are no longer of interest only for compiler optimisations but also for checking whether certain security conditions are met by programs [7, 5]. Type systems are being considered as static tools which can capture information about the security classes of values and which can be used to enforce predefined security policies [26, 12, 18, 17, 15, 23, 8, 13].

A wide range of languages have been subjected to study in the context of type-based approaches to security. These include imperative languages such as the ones considered in [28, 26, 24, 21, 17] and functional languages such as the Secure Lambda Calculus (SLam) [12], an extension of the λ-calculus suitable for trust analysis [18] and the Dependency Core Calculus [2]. Type systems for enforcing security properties in concurrent and mobile systems have also been studied in the framework of process calculi of the π-calculus family [13, 9], the ambient calculus [8] and the spi-calculus which is an extension of the π-calculus with cryptographic primitives [3].

The most closely related work to ours in the literature is the one on the SLam calculus [12]. Confined-λ resembles the SLam calculus through being a higher-order functional language. However, it differs from the SLam calculus considerably; Confined-λ supports distributed computing based on message passing in a multi-site system while the concurrent extension of the SLam calculus adopts a communication model based on shared memory. In investigating secure information flow properties for a language such as Confined-λ, one needs to consider possible sources of information flow which arise from the mobility of functions and channels between remote sites. This is a point which has not been addressed before within the framework of functional languages.

In Section 2, we state our assumptions about the underlying system and introduce the concept of mobility region in a more formal setting. Section 3 includes the definition of the syntax and the operational semantics of Confined-λ. In Section 4, we present a number of simple examples which illustrate the need for a semantic checking phase for Confined-λ to ensure that region violations do not occur. The type system which constitutes the basis of this semantic checking phase is discussed in detail in Section 5. We also prove in this section that well-typed programs enjoy the property of confinement in a mobility region. In Section 6 we have a closer look at the related body of work to clarify the contribution made by this paper. Section 7 includes ideas for future work.

2 Computing with mobility regions

The definition of a secrecy property, such as confinement in a mobility region, requires a clear statement of what is assumed about the underlying system and what is assumed to be observable in that system. Section 2.2 addresses these issues. We first discuss briefly what we mean by secrecy in the context of Confined-λ where all values are potentially mobile.

2.1 Secrecy and mobility

The notion of secrecy for a basic value such as an integer is quite straightforward; only a designated group of parties should be allowed to observe that integer. For example, if an integer represents a pin number for accessing a particular account, its mobility should be restricted to a part of the system which harbours as observers only the authorised users of that account. In a language such as Confined-λ a natural way to realize this would be to associate a group of users with the integer upon its creation.

The notion of secrecy for functions is slightly more elusive than those for basic values because functions are not passive values which are passed around. They abstract a behaviour which becomes activated when the function is applied. Hence it is reasonable to tie the notion of secrecy for functions to the actions they perform, the resources of the system they access and the results they return. We take the view that the appropriate secrecy level for a particular function would be best judged by the author of that function. For example, a function may be accessing sensitive resources of the system or it may be using the resources of the system intensively to perform its task. The author may then want to restrict the use of this function to a particular group of users which are trusted to use it in a responsible way. Likewise, it would be desirable to restrict the use of a function which returns private information about a particular user to a part of the system in which all of the observers are eligible to obtain this information.

Channels are instrumental in communicating information within the system. Acquiring a channel implies acquiring the right to input or output on that channel. The notion of secrecy for a channel coincides with that of the values which it can communicate.

2.2 System model

The language Confined-λ supports computation in a multi-site system in which each site has a unique name drawn from a finite set of site names $S = \{s_i \mid 1 \leq i \leq \text{size of the system}\}$. Each site is controlled by a particular user who both provides the code and observes the results of the computation at that site. This implies that a site name also identifies a user; the observation of a value at a particular site or the observation of a value by a particular user are essentially the same concepts. The same holds for the origination of code at a particular site and the origination of code by a particular user.

A program consists of multiple expressions, one for each site participating in the computation. Each expression is type-checked by a trusted Confined-λ compiler and the program is allowed to execute only if all of the expressions are well-typed.

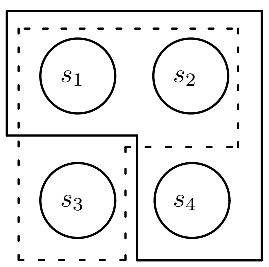
Observables We say that a value v is observable at a given site s, if the computation returns v at site s, or v appears at site s as an intermediate step in the computation. According to this definition, a value v could be observable at site s through being received over a channel at any point in the computation or through being contained in the code of a function which is received at s.

2.3 Mobility regions

As we discussed in the introduction, our approach is to give users full control over the flow of values which are created by the code written by them. Let us now consider a system which consists of just four sites with the names given in the set $S = \{s_1, s_2, s_3, s_4\}$ and suppose that the user at site s_1 provides the code for a service F which requires extensive processing power and s_4 is known to be equipped with a less powerful processor. It is reasonable for s_1 to restrict the use of F only to s_2 and s_3 besides itself. On the other hand, a service G may be carrying out a simple operation which does not demand much processing power but may return a value which the user at s_1 wants to keep secret from s_3. In this case, G should not be allowed to run at s_3.

We define a mobility region as a non-empty subset of the sites in the system. If we consider the above system with site names drawn from S, any non-empty subset of S is a valid mobility region. From the security perspective, mobility regions are used to declare a web of trust among a group of sites. Information which is required to be confined to a particular mobility region $r = \{s_1, s_2, s_3\}$ may be observed at any site within r but it should not be observed at any site outside r.

Continuing this example, $M_1 = \{s_1, s_2, s_3\}$ and $M_2 = \{s_1, s_2, s_4\}$ are the mobility regions which would be assigned to services F and G respectively. F can visit s_1, s_2, s_3 but not s_4. G can visit s_1, s_2, s_4 but not s_3.



- - - - - - mobility region for F

——— mobility region for G

Ordering on mobility regions Mobility regions can be interpreted as secrecy levels where the subset relation on mobility regions gives rise to an ordering on secrecy levels. Suppose that $\sqsubseteq$ is an ordering on secrecy levels where we write $A \sqsubseteq B$ if B indicates a higher secrecy level than A. Then the statements $r_1 \sqsubseteq r_2$ and $r_2 \subseteq r_1$ are equivalent. Note that by using mobility regions it would be possible to express as many secrecy levels as there are non-empty subsets of the site names in a system.

3 Confined-λ

The constants of the language consist of unit $()$, integers n and channel identifiers k. Basic constants, primitive operations on integers, variables, functions and function applications constitute the functional core of the language. This core is extended with channel allocation expressions which are used to allocate a channel of a certain data type, and expressions for sending and receiving data over channels. Canonical expressions and channel allocation expressions are labelled with mobility regions. The intuition behind this is that the values of these expressions are required to be confined to the region indicated in the label.

$$
\begin{array}{llcl}
\text{site identifier} & s & ::= & s_1 \mid s_2 \mid \ldots \\
\text{mobility region} & r & ::= & \{s\} \mid r \cup r \\
\text{type} & t & ::= & \texttt{unit} \mid \texttt{int} \\
 & & & \texttt{chan}(t) \mid t \rightarrow t \\
\text{constant} & c & ::= & ()^r \mid n^r \mid k^r \\
\text{expression} & e & ::= & c \mid x \mid \texttt{fn}^r\, x \Rightarrow e \\
 & & & \mid e_1\, e_2 \mid e_1\, \texttt{op}\, e_2 \\
 & & & \mid \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \\
 & & & \mid \texttt{chan}^r(t) \\
 & & & \mid e_1? \mid e_1\,!\,e_2
\end{array}
$$

3.1 Dynamic semantics

We define a small-step operational semantics for closed expressions of the language Confined-λ. The labels are included in the semantics merely to facilitate the proof of the properties we state in the following sections of the paper. They do not play any role in the evaluation of the expressions. Once an expression is deemed to be safe statically, it can be evaluated without any dynamic checks.

Values

Canonical expressions, that is the expressions which cannot be further evaluated, denote values of the language. We assume that a channel identifier is represented by a tuple which includes the identifier of the site it is created at and an integer which is freshly generated each time a new channel is allocated at a site. The expression $v_1 \,\mathtt{op}\, v_2$ denotes a value constructed from values v_1 and v_2 by using the primitive operation op.

$$\text{value} \qquad v \quad ::= \quad c \mid \mathtt{fn}^r\, x \Rightarrow e \mid v_1 \,\mathtt{op}\, v_2$$

Definition 1 (L_d). Function L_d is defined on values of the language. It yields the label of a given value which denotes its mobility region.

$$L_d(()^r) = r \qquad L_d(n^r) = r \qquad L_d(k^r) = r$$
$$L_d(\mathtt{fn}^r\, x \Rightarrow e) = r \qquad L_d(v_1 \;\mathtt{op}\; v_2) = L_d(v_1) \cap L_d(v_2)$$

Evaluation rules

The evaluation rules are defined by using transitions between system states written as $CI, P \longrightarrow CI', P'$. A system state is represented by a channel identifier set CI and a process pool P in the style of [6]. The channel identifier set contains the region-annotated channel identifiers created so far in the computation and the process pool is a set of site names and expression pairs. A pair written as $[s : e]$ indicates that expression e is to be executed at site s. We use the notation $P[s : e]$ to denote the process pool $P \cup [s : e]$.

The evaluation rules are presented in Figure 1. The rules for the sequential subset correspond to standard evaluation rules for call-by-value functional languages put in a distributed context. The notation $e[v/x]$ denotes the substitution of v for x in e where the appropriate renaming has been performed to avoid the capturing of free variables. The rule for channel allocation states that a fresh channel identifier is generated upon the execution of a channel allocation expression. The rule for communication shows how a value is transmitted from one site to another as a result of synchronisation over a channel. Note that the transmitted value can be a basic value, a channel or a function. The possibility of transmitting channels and functions allows the connectivity between remote sites and the interaction capabilities of expressions to evolve dynamically.

4 Examples

In Sections 1 and 2 we introduced mobility regions as a useful device for programmers to specify how they wish to constrain the flow of values. In the next section we will present a set of typing rules, which forces the values to be confined to their specified mobility regions. These typing rules prevent region violations where we use the phrase *region violation* to refer to the observability of a value outside its specified mobility region. Our aim here is to give the intuition behind our typing rules by illustrating major sources of region violations by means of simple examples.

Example 1: Uncontrolled use of channels

In the following example, *A* is the creation site for two values: channel chAC intended for communication between sites *A* and *C* and function f which is allowed to roam freely. The function f requires a channel as an argument. It uses this channel to input a value and forwards the input value back to site *A* by using the channel chAC in its closure. When executed, the expression at *A* will send f to *C* on a previously allocated channel chTop which is known to both *A* and *C*. It will then start listening on chAC. We use ; here as a shorthand for sequencing. It can easily be encoded in the syntax of Confined-λ.

On the other hand, the execution of the expression at *B* will cause an integer to be computed and sent to *C* over chBC. Note that we assume chBC to be a channel with mobility region $\{B, C\}$. The annotation of the created value shows us that the author of the code at *B* wishes to restrict the observability of this value to *B* and *C*. This wish, however, will not come true as the application of f to chBC at *C*, causes secretBC to be received at *C* and subsequently be sent to *A*.

```
A:  let   chAC = chan^{A,C}(int)
    in    let f = fn^{A,B,C} ch ⇒
                    let x = (ch?)
                    in chAC ! x
          in chTop !  f;
             (chAC ?)

B:  let secretBC =  ... ^{B,C}
    in chBC ! secretBC

C:  let f′ = (chTop ?)
    in f′ chBC
```

Our approach in preventing violations such as this is to require the mobility region of a channel and that of the value it communicates to be identical. In our example, this would prevent secretBC from being sent over chAC.

Example 2: Escaping values

In this example, *A* is the origination site for function f which will be sent to *B* when the expression is executed. However, when applied at *B*, f will return a value which was originally intended to remain at *A*.

```
A:  let f = fn^{A,B}() ⇒  ... ^{A}
    in chAB ! f

B:  (chAB?)()
```

Function application

$$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : e_1\, e_2] \longrightarrow CI', P'[s : e_1'\, e_2]}$$

$$\frac{CI, P[s : e_2] \longrightarrow CI', P'[s : e_2']}{CI, P[s : v\, e_2] \longrightarrow CI', P'[s : v\, e_2']}$$

$$CI, P[s : (\mathtt{fn}^r\, x \Rightarrow e)\, v] \longrightarrow CI, P[s : e[v/x]]$$

Let binding

$$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2] \longrightarrow CI', P'[s : \mathtt{let}\, x = e_1'\, \mathtt{in}\, e_2]}$$

$$CI, P[s : \mathtt{let}\, x = v\, \mathtt{in}\, e_2] \longrightarrow CI, P[s : e_2[v/x]]$$

Arithmetic operation

$$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[e_1\, \mathsf{op}\, e_2] \longrightarrow CI', P'[s : e_1'\, \mathsf{op}\, e_2]}$$

$$\frac{CI, P[s : e_2] \longrightarrow CI', P'[s : e_2']}{CI, P[s : v\, \mathsf{op}\, e_2] \longrightarrow CI', P'[s : v\, \mathsf{op}\, e_2']}$$

Channel allocation

$$CI, P[s : \mathtt{chan}^r(t)] \longrightarrow CI \cup k^r, P[s : k^r] \;\text{ where } k^r \notin CI$$

Send

$$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : e_1\, !\, e_2] \longrightarrow CI', P'[s : e_1'\, !\, e_2]}$$

$$\frac{CI, P[s : e_2] \longrightarrow CI', P'[s : e_2']}{CI, P[s : k^r\, !\, e_2] \longrightarrow CI', P'[s : k^r\, !\, e_2']}$$

Receive

$$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : e_1 ?] \longrightarrow CI', P'[s : e_1' ?]}$$

Communication

$$CI, P[s_1 : k^r\, !\, v][s_2 : k^r ?] \longrightarrow CI, P[s_1 : ()^r][s_2 : v]$$

Figure 1. Evaluation Rules

Violations such as this can be prevented by requiring the mobility region of the result of a function to be at least as large as that of the function itself.

Example 3: Remotely created values

The function f below, which is allowed to be mobile between *A* and *B*, contains a channel allocation expression. The annotation shows us that this channel is intended to be used between *A* and *C*. However, when the function is applied at *B*, the channel will be created at *B*. According to our definition in Section 2.2, this will cause the values transmitted on this channel to be observable at *B*.

```
A:  let f = fn{A,B} () ⇒
              let chAC = chan{A,C}(int)
              in ...
    in chAB ! f

B:  (chAB?) ()
```

The problem here arises from the fact that functions are abstractions; the code enclosed by the function may be executed at a site which is different from the one where the function was created. Region violations due to this fact may be prevented by requiring the mobility region of the values created by the function to be a superset of all the sites where the function may be applied.

Example 4: Transmission within a closure

The function f below refers to the integer secretInt which is present in its definition environment. The annotation shows us that this integer is intended to remain at *A*. However, according to the dynamic semantics presented in Section 3.1, when the expression is executed secretInt will be substituted in the code of f and transmitted to *B*.

```
A:  let  secretInt =  ... {A}
    in   let f = fn{A,B} () ⇒...
                 (* do something *)
                 (* with secretInt *)
         in chAB ! f

B:  (chAB?) ()
```

To prevent region violations of this sort we need to address the dependency between the mobility region of a function and the values which occur in its definition environment.

5 Static semantics

The property called confinement in a mobility region was motivated in the preceding sections. The purpose of the type-system presented in this section is to enforce that property. According to our system model, each site participating in the distributed computation provides an expression which is well-formed according to the syntax presented in Section 3. This expression is then analysed to check whether it is well-formed with respect to the rules of the type-system given in Figure 3. We discuss these typing rules in detail in Section 5.2.

5.1 Objects of the type system

The types of our system are pairs. The first component of the pair is a type in the conventional sense whereas the second component is used to record the mobility region. A type environment is a finite map from variables to types and a channel environment is a finite map from channel identifers to types. Channel identifiers may occur in expressions just as constants of other types such as integers. One could think of them as identifiers of predefined channels. In the type system channel environments are used to look up the types of these predefined channels.

mobility region	r	::=	$\{s\} \mid r \cup r$
raw type	$\bar{\tau}$	::=	$\texttt{unit} \mid \texttt{int}$
			$\mid \texttt{chan}[\bar{\tau}]$
			$\mid \tau \rightarrow \tau$
annotated type	τ	::=	$(\bar{\tau}, r)$
type environment	Γ	::=	$[\,] \mid \Gamma[x \mapsto \tau]$
channel environment	CE	::=	$[\,] \mid CE[k^r \mapsto \tau]$

We use two forms of judgements in our type system. A judgement of the form $\vdash \tau$ indicates that type τ is well-formed according to the rules given in Figure 2. We say that a type environment and a channel environment are well-formed if all of the types in their range are well-formed. A judgement of the form $r, CE, \Gamma \vdash e : \tau$ indicates that the expression e can be assigned a well-formed type τ at any site within the mobility region r, by using the typing rules of Figure 3 and the well-formed environments CE and Γ. A distinctive point about our type system is the use of mobility regions in typing judgements along with typing and channel environments. This allows us to keep track of the static estimation of the origin and the execution site of an expression.

Definition 2 (L_s). L_s is a function used in typing rules to extract a mobility region from a type so that $L_s(\bar{\tau}, r) = r$.

$$\vdash (\texttt{unit}, r) \qquad \vdash (\texttt{int}, r)$$

$$\frac{\vdash (\bar{\tau}, r)}{\vdash (\texttt{chan}[\bar{\tau}], r)} \qquad \frac{\vdash \tau_1 \quad \vdash \tau_2 \quad r \subseteq L_s(\tau_2)}{\vdash (\tau_1 \rightarrow \tau_2, r)}$$

Figure 2. Well-formed Types

Definition 3 ($Type_{CE}$). $Type_{CE}$ is a family of functions indexed by channel environments. It is defined on constants of the language.

$$Type_{CE}(()^r) = (\texttt{unit}, r) \quad Type_{CE}(n^r) = (\texttt{int}, r)$$
$$Type_{CE}(k^r) = CE(k^r)$$

Definition 4 (Annotation erasure$\lfloor \cdot \rfloor$). We write $\lfloor \tau \rfloor$ for a simple type which is obtained from τ by erasing all annotations recursively. (e.g. $\lfloor (\texttt{int}, r) \rfloor = \texttt{int}$)

5.2 Typing rules

The typing rules are presented in Figure 3. Before looking at the rules in isolation, it would be helpful to note the following. The mobility region r used in the context of the judgement at the root of a typing derivation would be a singleton containing the name of the site at which the top-level expression is type-checked.

The role of the typing rule **(con)** for constants is to record in the type of a constant its specified mobility region. The side condition is used to ensure that no spurious declarations are made. For example s_1 cannot create a value and declare its mobility region to be s_2. The mobility region of a value is guaranteed to include its origination site.

The typing rule **(var)** is used for typing variables. The mobility region in the type of a variable is required to include the mobility region where the value of the variable may be found. This rule is crucial in preventing the kind of violation shown in Example 4; thanks to this rule we do not allow a value to appear in the closure of a function which may visit some sites where the value should not be observed. This rule also implies that a function cannot refer to a channel which was created outside its body unless this channel's mobility region is a superset of the mobility region of the function. This gives us a useful technical device for controlling the propagation of input output capabilities caused by the mobility of functions.

There are three important aspects of the typing rule **(fn)** for functions. Firstly, functions are canonical expressions just as special constants. Their typing follows the same reasoning; we record the specified mobility region of a function in its type. The second aspect is to do with the fact that a mobility region of a function represents the sites which the function is allowed to visit. The well-formedness condition for function types ensures that a value returned by the function is allowed to be observed in the mobility region of the function itself. Region violations such as the one presented in Example 2 would be prevented by the presence of this condition. Finally, we need to take into consideration that the body of the function may be evaluated in a context different from that of the function abstraction. All we know about it is that it will be one of the sites that the function is allowed to visit. We type the body of an expression with respect to the mobility region of the function. Example 3 illustrates the significance of this final aspect.

The rules for applications and let expressions are quite straightforward. The typing rule **(op)** for primitive operations indicates that an operation op is allowed on integers only. The mobility region of an integer constructed from two integers using this operation is required to be the intersection of the regions of the components. Otherwise, a value could appear at a site which is outside its region through being a part of the composed value.

The concept of mobility region for a channel is not different from that of other data types. It represents the sites at which a channel is allowed to appear. The typing rule **(chan)** for channel allocation expressions ensures that a channel with a particular mobility region is allowed to be created at one of the sites in its mobility region.

The single most important point about the typing rules **(send)** and **(receive)** is that they force channels to carry values of the same type as themselves. We have motivated this rule by Example 1.

5.3 Confinement in a mobility region

We now state some properties of our system which will eventually lead us to formalise and prove the confinement in a mobility region property. Our type system guarantees that if an expression is well-typed at a particular site, it enjoys this property and therefore can be safely run at that site. We refer the interested readers to the Appendix for the details of the proofs.

Theorem 1 below shows that the type of an expression is preserved under transitions of the system.

Theorem 1 (Subject reduction). Assume that

- $CI, P[s : e] \longrightarrow CI', P'[s : e']$
- $CI = Dom(CE)$
- $r, CE, [\,] \vdash e : \tau$.

Then there exists a CE' such that $r, CE', [\,] \vdash e' : \tau$ and $CI' = Dom(CE')$.

Proof. The proof is given by straightforward induction on the depth of inference of $CI, P[s : e] \longrightarrow CI', P'[s : e']$

$$\textbf{(con)} \quad \frac{Type_{CE}(c) = \tau \text{ where } \tau = (\bar{\tau}, r) \quad r' \subseteq r}{r', CE, \Gamma \vdash c : \tau}$$

$$\textbf{(var)} \quad \frac{\Gamma(x) = \tau \text{ where } \tau = (\bar{\tau}, r) \quad r' \subseteq r}{r', CE, \Gamma \vdash x : \tau}$$

$$\textbf{(fn)} \quad \frac{r, CE, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \quad r' \subseteq r}{r', CE, \Gamma \vdash \texttt{fn}^r\ x \Rightarrow e : (\tau_1 \rightarrow \tau_2, r)}$$

$$\textbf{(app)} \quad \frac{r', CE, \Gamma \vdash e_1 : (\tau \rightarrow \tau_1, r) \quad r', CE, \Gamma \vdash e_2 : \tau}{r', CE, \Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\textbf{(let)} \quad \frac{r', CE, \Gamma \vdash e_1 : \tau \quad r', CE, \Gamma[x \mapsto \tau] \vdash e_2 : \tau_2}{r', CE, \Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2}$$

$$\textbf{(op)} \quad \frac{r', CE, \Gamma \vdash e_1 : (\texttt{int}, r_1) \quad r', CE, \Gamma \vdash e_2 : (\texttt{int}, r_2)}{r', CE, \Gamma \vdash e_1\, \texttt{op}\, e_2 : (\texttt{int}, r_1 \cap r_2)}$$

$$\textbf{(chan)} \quad \frac{t = \lfloor(\bar{\tau}, r'')\rfloor \quad r' \subseteq r}{r', CE, \Gamma \vdash \texttt{chan}^r(t) : (\texttt{chan}[\bar{\tau}], r)}$$

$$\textbf{(send)} \quad \frac{r', CE, \Gamma \vdash e_1 : (\texttt{chan}[\bar{\tau}], r) \quad r', CE, \Gamma \vdash e_2 : (\bar{\tau}, r)}{r', CE, \Gamma \vdash e_1!e_2 : (\texttt{unit}, r)}$$

$$\textbf{(receive)} \quad \frac{r', CE, \Gamma \vdash e : (\texttt{chan}[\bar{\tau}], r)}{r', CE, \Gamma \vdash e? : (\bar{\tau}, r)}$$

Figure 3. Typing Rules

by considering the possible forms of e. The cases referring to the third evaluation rule for applications and the second evaluation rule for let bindings appeal to Lemma 1. The case for communication makes use of the fact that a value has a unique type within its mobility region, regardless of the site at which the typing takes place. This is formalised by Lemma 2 below. □

Lemma 1 (Substitution). If $r, CE, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2$ and $r, CE, \Gamma \vdash v : \tau_1$ then $r, CE, \Gamma \vdash e[v/x] : \tau_2$.

Lemma 2 (Preservation of typability within a region). If $r, CE, \Gamma \vdash v : (\bar{\tau}, r')$ then for any r'' such that $r'' \subseteq r'$ it is the case that $r'', CE, \Gamma \vdash v : (\bar{\tau}, r')$.

If expression e is well-typed as site s and it is evaluated at site s returning the value v as its result, then s is guaranteed to be within the mobility region specified for value v. Theorem 2 is the main result which we present in this paper. We use the symbol $\longrightarrow^*$ below to represent a sequence of transitions.

Theorem 2 (Confinement in a mobility region). Assume for a closed e that

- $\{s\}, CE, [\,] \vdash e : \tau$
- $CI = Dom(CE)$
- $CI, P[s : e] \longrightarrow^* CI', P'[s : v]$.

Then $\{s\} \subseteq L_d(v)$.

Proof. The proof of the above theorem follows from Theorem 1 and Lemmas 3 and 4. Lemma 3 below shows that the type of an expression conservatively estimates the sites where the expression would be accepted as a result of the type-checking phase. If an expression e evaluates to a value v as is assumed in our theorem, by Theorem 1 and Lemma 3 we know that the label in the type of v will contain $\{s\}$. We also know by Lemma 4 that the mobility region annotations in the type of a value are consistent with the annotations provided as a part of the syntax, i.e. $L_d(v) = L_s(\tau)$. This allows us to conclude that $\{s\} \subseteq L_d(v)$. □

Lemma 3 (Conservative estimation). If $r, CE, \Gamma \vdash e : \tau$ then $r \subseteq L_s(\tau)$.

Lemma 4 (Consistency of labels). If $r, CE, \Gamma \vdash v \ : \ \tau$ then $L_d(v) = L_s(\tau)$.

6 Related work

In the introduction we pointed to several streams of work on security which build on findings and methods of programming language research. There is a large body of work on type systems for security which is continuing to grow. We present here an in-depth discussion about the works which we consider as closely related to ours.

The work on Confined-λ presented in this paper has developed as part of a general study of mobile computation with functions and support for security in that context. An important concern has been to maintain the links between our work and functional programming languages such as Concurrent ML and Facile which have working implementations. We have aimed at designing type systems which are based on simple notions of security and easy-to-explain principles so that they can eventually become of practical interest as extensions to the existing type systems of these languages.

The SLam calculus The purely functional subset of the SLam calculus and Confined-λ are closely related. In both languages, values are annotated to indicate their secrecy levels. In Confined-λ we call them mobility regions to make explicit that these annotations are used to restrict the mobility of values. However, there would be no essential difference if we interpreted them as sets of users who are allowed to observe these values as suggested by the SLam calculus.

The annotations of the SLam calculus, called security properties, are more expressive than the annotations of Confined-λ. This is mainly due to the differences in our motivations. The SLam calculus aims at capturing refined notions of security by exploiting rich type structures. Taking a complementary approach, the aim of this paper is to establish a simple notion of secrecy for distributed computation by remaining close in spirit to conventional type systems for functional languages.

The work on the SLam calculus distinguishes between direct readers and indirect readers of values in order to be able to address indirect information flows such as those caused by branching on high security vales in conditional expressions. The annotations of the SLam calculus values are tuples, where the first and the second components specify respectively its authorised direct readers and indirect readers. The authors also introduce dual notions to readers and indirect readers; creators and indirect creators represent users who might have created the objects directly or indirectly. Mobility region annotations of Confined-λ can be likened to reader annotations of the SLam calculus with no distinction between direct readers and indirect readers.

A formal study of the correspondence between the functional subsets of the SLam calculus restricted to reader annotations and Confined-λ with mobility region annotations could be a future work in its own right. We now discuss the typing of a function abstraction and application in both systems to illustrate the similarities and differences of the adopted approaches.

The following is the typing rule for functions in the SLam calculus. The annotation κ is a tuple of the form (r, ir) where r is the direct reader annotation and ir is the indirect reader annotation. A global condition on the type system requires r to indicate a higher secrecy level than ir.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e : \tau_2)_\kappa : (\tau_1 \rightarrow \tau_2, \kappa)}$$

A function which is typed with respect to the rule above can only be applied by those readers whose secrecy level is at least as high as the one indicated by the reader annotation r of the function. In order to enforce this, application expressions are annotated with the security levels of programmers and a side condition is used in the typing rule to check that the programmer is authorised to apply the function.

$$\frac{\Gamma \vdash e_1 : (\tau_1 \rightarrow \tau_2, (r, ir)) \quad \Gamma \vdash e_2 : \tau_1 \quad r \sqsubseteq r'}{\Gamma \vdash (e_1\, e_2)_{r'} : \tau_2 \bullet ir}$$

It is the responsibility of the trusted compiler to ensure that programmers provide annotations which are consistent with their actual security levels. The rule above also shows that the result of the application is to be protected at the security level $\tau_2 \bullet \kappa$. However, this is not relevant to our present discussion.

We now look at the typing rules for functions and applications in Confined-λ. Note that we have omitted the channel environment CE in the typing judgements as we are comparing the sequential subsets only. In order to facilitate the comparison of the two systems we could read a judgement of the form $r, \Gamma \vdash e : \tau$ as follows: an expression e provided by an author who is a member of r, can be assigned type τ.

$$\frac{r, \Gamma[x \ \mapsto \ \tau_1] \vdash e : \tau_2 \quad r' \subseteq r}{r', \Gamma \vdash \mathtt{fn}^{r}\, x \Rightarrow e : (\tau_1 \rightarrow \tau_2, r)}$$

Informally speaking, the role played by the typing rule for applications in the SLam calculus is assigned to the typing rule for functions in our type system. The only essential difference is that we force the annotation of the function to include its author. This reflects our view that it would be natural to allow the use of a function by its own author. The body of the function above can be executed by the users in r, hence its typing takes place in a context which includes r. This is how we enforce authorised access to the code of the function. Our approach eliminates the need for annotating application expressions as in SLam. The typing rule for applications in Confined-λ is as follows:

$$\frac{r', \Gamma \vdash e_1 : (\tau \rightarrow \tau_1, r) \quad r', \Gamma \vdash e_2 : \tau}{r', \Gamma \vdash e_1\, e_2 : \tau_1}$$

As we have discussed in the introduction, the SLam calculus and Confined-λ adopt different communication models. The SLam calculus assumes a shared memory where the communication is based on mutable data structures. Confined-λ, however, supports communication by message passing as this is a more natural method of communication in distributed systems. We also note that communication based on shared memory is inherently asynchronous whereas the sender of a message and its receiver must synchronise in Confined-λ. These differences preclude a direct comparison of the two type systems as we were able to do for the functional core. It would be interesting to formalise an extension of the SLam calculus with distributed shared memory and devise a type system for that language which enjoys the secrecy property of this paper. We would regard any similarity between this type system and that of Confined-λ as an encouraging sign that secrecy can be dealt with using similar techniques in functional languages which adopt different communication models.

Process calculi The channel-based communication model and the presence of explicit localities relate Confined-λ to process calculi for concurrent and distributed computation, particularly to higher-order extensions of the π-calculus such as the ones presented in [29, 25]. We are not aware of any other work on these calculi which focuses on a notion of secrecy such as ours. On the other hand, secrecy by typing has been investigated in the framework of process calculi whose relation to Confined-λ is relatively indirect. For example, a type system which is aimed at detecting information leaks to the environment has been developed for the spi-calculus [3]. This work provides insights into the questions of formulating and proving secrecy properties for concurrent systems where interaction capabilities evolve dynamically. However, the languages Confined-λ and the spi-calculus differ in several respects. The spi-calculus is a calculus which is particularly suited for the description and analysis of cryptographic protocols. It extends the core of the π-calculus with cryptographic primitives. There is no direct way of expressing the communication of functions between processes. The characteristic feature of Confined-λ, however, is that it offers a simple model for distributed computation which is based on the mobility of functions between concurrently executing expressions at remote sites.

A language which is unlike all of the other languages cited above but which can be regarded as related to Confined-λ is uPLAN [14]. This language has been defined to provide a formal model of computation in active networks. It assumes a fixed set of sites and the computation is based on the mobility of functions. In these respects it is similar to Confined-λ. However, the mobility in uPLAN is based on a facility for remote evaluation of functions at explicitly specified locations. The authors propose a notion of secrecy inspired by the spi-calculus. The proof of their secrecy property does not depend on a type-based approach such as ours.

7 Conclusions and future work

We have presented a higher-order functional language which supports distributed computing by allowing expressions at different sites to communicate through channels. In this language, authors of code can declare whether they wish particular values to flow only within a particular subsystem, so that only the users in that subsystem can observe these values. We have presented a type system to guarantee that if the program is well-typed the values will indeed remain in that subsystem. The concept of mobility region has been introduced as a useful device for achieving this.

The issues related to information flow due to covert storage channels or covered timing channels are left out of the scope of this paper. We are planning to investigate these issues for an extension of Confined-λ with conditional expressions. The analogy between secrecy levels and mobility regions should allow us to benefit from the works which focus primarily on noninterference properties such as the ones studied in [12, 27, 19] and which focus on timing leaks [4].

In the process calculi framework, some researchers have exploited type systems to enforce locality conditions on the use of capabilities [29, 22]. It is possible to regard confinement in a mobility region as a locality condition; we restrict the distribution of communication of capabilities to a part of the system as a means of controlling the flow of information within the system. Distinguishing between input and output capabilities, as in [29, 22], could allow us to explore more refined notions of secure information flow. This remains as an interesting direction for future work.

Acknowledgements: I would like to thank Stephen Gilmore for our useful discussions in the process of writing this paper. I also would like to thank the referees for their helpful comments. My research is partially funded by the EPSRC grant R35414.

References

[1] M. Abadi. Secrecy in programming-language semantics. In *Electronic Notes in Theoretical Computer Science*, volume 20. Elsevier Science Publishers, 2000.

[2] M. Abadi, A. Banarjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 147–160. ACM Press, 1999.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[4] J. Agat. Transforming out time leaks. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 40–53, Boston, Massachusetts, USA, 2000. ACM Press.

[5] J. Banâtre, J. Bryce, and D. L. Métayer. Compile-time detection of information flow in sequential programs. In *Proceedings of Third European Symposium on Research in Computer Security*, volume 875 of *LNCS*, pages 55–73, Brighton, UK, 1994. Springer-Verlag.

[6] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 119–129. ACM Press, 1992.

[7] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis of processes for no read-up and no write-down. In *Proceedings of FOSSACS'99*, number 1578 in Lecture Notes in Computer Science, pages 120–134. Springer-Verlag, 1999.

[8] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of International Colloquim on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 230–239. Springer, 1999.

[9] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*, volume 1877 of *LNCS*, pages 365–379. Springer, 2000.

[10] A. Fugetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[11] 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.

[12] 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.

[13] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proceedings of International Colloquim on Automata, Languages and Programming (ICALP)*, pages 415–427, Geneva, Switzerland, 2000. Springer-Verlag.

[14] P. Kakkar, C. A.Gunter, and M. Abadi. Reasoning About Secrecy for Active Networks. In *Proceedings of the Computer Security Foundations Workshop (CSFW)*, 2000.

[15] 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.

[16] R. Milner, M. Tofte, R.Harper, and D.MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

[17] A. C. Myers and B. Liskov. JFlow: Practical mostly-static information flow control. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, 1999.

[18] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.

[19] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 46–57, Montreal,Canada, 2000.

[20] J. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.

[21] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[22] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of International Colloquim on Automata, Languages and Programming (ICALP)*, volume 1443 of *LNCS*. Springer Verlag, 1998.

[23] C. Skalka and S. Smith. Static enforcement of security with types. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, Montreal, Canada, 2000.

[24] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, San Diego, CA, USA, 1998.

[25] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College, London University, 1989.

[26] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of Theory and Practice of Software Development (TAPSOFT)*, pages 607–621, Lille, France, 1997.

[27] D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, 1998.

[28] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow anaylsis. *Journal of Computer Security*, 4(3):1–21, 1996.

[29] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*, pages 557–572, Santa Barbara, US, 1999. Springer-Verlag.

Appendix

Lemma 1: Substitution *If* $r, CE, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2$ *and* $r, CE, \Gamma \vdash v : \tau_1$
then $r, CE, \Gamma \vdash e[v/x] : \tau_2$.

Proof: The proof is given by induction on the typing derivation of
$r, CE, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2$. Some representative cases are as follows:

- **(var)** $r, CE, \Gamma[x \mapsto \tau_1] \vdash x' : \tau_2$.

i. $x \neq x'$. This implies that x' is in the domain of Γ such that $\Gamma(x') = \tau_2$ satisfying the necessary conditions of the typing rule **(var)**. We know that $x'[v/x] = x'$. We can conclude by the typing rule **(var)** that $r, CE, \Gamma \vdash x' : \tau_2$.

ii. $x = x'$. This implies that $\tau_2 = \tau_1$. Since $x[v/x] = v$ we need to establish $r, CE, \Gamma \vdash v : \tau_2$. This is immediate from the assumptions.

- **(fn)** $r', CE, \Gamma[x \mapsto \tau_1] \vdash \texttt{fn}^r\, y \Rightarrow e : (\tau_2 \rightarrow \tau_3, r)$ where $r, CE, \Gamma[x \mapsto \tau_1][y \mapsto \tau_2] \vdash e : \tau_3$ and $r' \subseteq r$ and $x \neq y$. By induction hypothesis on e we have $r, CE, \Gamma[y \mapsto \tau_2] \vdash e[v/x] : \tau_3$. We get the required result by applying the typing rule **(fn)**. □

Lemma 2: Preservation of typability within a region
If $r, CE, \Gamma \vdash v : (\bar{\tau}, r')$ *then for any* r'' *such that* $r'' \subseteq r'$ *it is the case that* $r'', CE, \Gamma \vdash v : (\bar{\tau}, r')$.

Proof: The proof is given by induction on the typing derivation of
$r, CE, \Gamma \vdash v : (\bar{\tau}, r')$.

- **(con)** $r, CE, \Gamma \vdash c : (\bar{\tau}, r')$ where $Type_{CE}(c) = (\bar{\tau}, r')$ and $r \subseteq r'$. For any $r'' \subseteq r'$ the side conditions would still hold for establishing $r'', CE, \Gamma \vdash c : (\bar{\tau}, r')$.

- **(fn)** Similar to the above case.

- **(op)** $r, CE, \Gamma \vdash v_1 \;\texttt{op}\; v_2 : (\texttt{int}, r')$ where $r, CE, \Gamma \vdash v_1 : (\texttt{int}, r_1)$ and $r, CE, \Gamma \vdash v_2 : (\texttt{int}, r_2)$ and $r' = r_1 \cap r_2$. By induction hypothesis on v_1 and v_2 we have that $r'', CE, \Gamma \vdash : v_1 : (\texttt{int}, r_1)$ and $r'', CE, \Gamma \vdash v_2 : (\texttt{int}, r_2)$. The required result follows from the typing rule **(op)**. □

Theorem 1 (Subject Reduction)
Assume $CI, P[s : e] \longrightarrow CI', P'[s : e']$ *and* $CI = Dom(CE)$ *and* $r, CE, [\,] \vdash e : \tau$. *Then there exists a* CE' *such that* $r, CE', [\,] \vdash e' : \tau$ *and* $CI' = Dom(CE')$.

Proof: We have outlined the proof in Section 5.3. We include here the proofs for some of the representative cases only.

- **(let-2)** $CI, P[s : \texttt{let}\, x = v\, \texttt{in}\, e_2] \longrightarrow CI, P[s : e_2[v/x]]$ and
$r, CE, [\,] \vdash \texttt{let}\, x = v\, \texttt{in}\, e_2 : \tau_2$ where
$r, CE, [\,] \vdash v : \tau$ and
$r, CE, [x \mapsto \tau] \vdash e_2 : \tau_2$. By Lemma 1 we establish the required result.

- **(com)** $CI, P[s_1 : k^r!v][s_2 : v] \longrightarrow CI', P'[s_1 : ()^r][s_2 : v]$ and $CI = Dom(CE)$ and $\{s_1\}, CE, \Gamma \vdash k^r!v : (\texttt{unit}, r)$ and $\{s_2\}, CE, \Gamma \vdash k^r? : (\bar{\tau}, r)$. The typing rules for expressions at s_1 and s_2 require that $\{s_1\}, CE, \Gamma \vdash k^r : (\texttt{chan}[\bar{\tau}], r)$ where $\{s_1\} \subseteq r$ and $\{s_1\}, CE, \Gamma \vdash v : (\bar{\tau}, r)$ and $\{s_2\}, CE, \Gamma \vdash k^r : (\texttt{chan}[\bar{\tau}], r)$ where $\{s_2\} \subseteq r$. It is obvious that $\{s_1\}, CE, \Gamma \vdash ()^r : (\texttt{unit}, r)$. By Lemma 2, using $\{s_1\}, CE, \Gamma \vdash v : (\bar{\tau}, r)$ and $\{s_2\} \subseteq r$ as assumptions, we can establish $\{s_1\}, CE, \Gamma \vdash v : (\bar{\tau}, r)$ which is the required result. □

Lemma 3 (Conservative estimation) *If* $r, CE, \Gamma \vdash e : \tau$ *then* $r \subseteq L_s(\tau)$.
Proof: The proof is given by induction on the typing derivation of $r, CE, \Gamma \vdash e : \tau$. Some representative cases are as follows:

- The proofs for cases **(con)**,**(var)** and **(fn)** follow immediately from the side conditions imposed by the typing rules.

- **(app)** $r, CE, \Gamma \vdash e_1 e_2 : \tau$. The typing rule **(app)** requires that
$r, CE, \Gamma \vdash e_1 : (\tau \rightarrow \tau_1, r')$ and $r, CE, \Gamma \vdash e_2 : \tau$. By the induction hypothesis on e_1 we have that $r \subseteq r'$. It follows from the well-formedness condition for function types that $r' \subseteq L_s(\tau_1)$. By transitivity of the subset relation $\subseteq$ we conclude that $r \subseteq L_s(\tau_1)$. □

Lemma 4 (Consistency of labels)
If $r, CE, \Gamma \vdash v : \tau$ *then* $L_d(v) = L_s(\tau)$.

Proof: A simple inspection of the typing rules is sufficient to prove this lemma. □