A Polymorphic Type and Effect System for Detecting Mobile Functions

Z. Dilsun Kırlı*

Abstract

We introduce the language Mobile-λ which supports mobile computation by combining a functional core with constructs for transmitting and receiving values across remote sites. Mobile agents have a natural representation as function closures in Mobile-λ. We develop a static type and effect system where the types are annotated to reveal the flow of functions and channels. We present an analysis of types which conservatively estimates the potentially mobile functions and channels. The paper also includes the specification of the dynamic semantics of Mobile-λ in relational style, proofs for the consistency of static and dynamic semantics and the soundness of our analysis.
Keywords Functional languages, mobile computation, program analysis, type and effect systems, Facile

1 Introduction

Functional computation has some characteristics which make it promising to adopt a functional language as the core of a mobile computation language. One such characteristic is that functions are first-class values. They can flow as arguments to other functions or they can be returned as results. In a setting where their communication between different sites is facilitated, functions become a natural candidate for representing mobile agents since they are code-containing objects capable of enhancing the dynamic behaviour of a system. It is also the case that expressive and well-understood type systems have been defined for functional languages. In a setting where functions can be viewed as mobile agents, this fact implies that one can express a wide

*Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, UK. E-mail: zdk@dcs.ed.ac.uk., tel:+44 131 650 48 89

range of agents and yet be able to reason about their behaviour by exploiting the type system of the language in which they are written.

Facile [29, 6] is a language which encompasses the above mentioned characteristics. It is based on a well defined computational model which combines Standard ML [16] with a model of concurrency based on CCS [14] and its higher-order mobile extensions such as CHOCS [27] and the π-calculus [15]. It facilitates the transmission of first-class values including channels and functions over typed channels. Therefore, it is an apt choice to consider a Facile-like language when focusing on a particular problem relevant to mobile computation with mobile functions. In this paper, we introduce such a language called Mobile-λ and we focus on the problem of statically estimating potentially mobile functions and channels in the framework of Mobile-λ.

The estimation of mobile entities can be useful for compiler optimisations. We can provide supporting evidence for this argument by considering the implementation of the Facile language by Knabe [11] which facilitates the transmission of mobile agents across a network of heterogeneous nodes. In this language, users are required to provide annotations to identify potentially mobile functions so that the compiler generates a standard transmissible representation for these functions. The code for the functions which are immobile can be optimised for the local machine and the compiler need not generate and store a standard transmissible representation for them. A static analysis which estimates potentially mobile functions would make it possible to automate the identification of mobile values relieving the programmer from having to provide annotations and reducing the risks of errors due to omitted annotations.

Another area where information with respect to mobile entities can be put to use is providing a profile of a program so that programmers can statically reason about the transmission overhead their programs are likely to incur. In Facile and most higher-order languages, a function is compiled into a function closure which contains the code of the function together with the bindings from the definition environment of the function. According to the data space management policy employed by the implementation of Facile, the transmission of a function occurs by copying its closure to the destination site. Channels in Facile reside on the node where they were created. Their transmission requires that a network reference to them be established and maintained. Therefore, a profiler which estimates mobile functions and channels can be considered to estimate the cost of copying closures and network references. The latter can be particularly useful in that minimising network references is a key motivation for adopting mobile computation.

The problem of detecting mobile functions and channels is closely related to detecting the flow of control from one program point to another. So far we have considered the higher-order features in a positive light due to their expressive power. However, their presence poses challenges in detecting the flow of control. The literature contains a wide range of analyses which have been devised to approximate the functions which can be called from a particular point in a program [23, 5, 17, 8]. Some authors have pointed out the intuitive connection between reasoning about types and control flow in higher-order languages in the sense that they both derive invariants about the potential bindings of variables in a program. Research has been carried out in extending control flow systems to perform type analyses [21] and in extending type systems to perform control flow analyses [26]. Another direction of research has focused on systematic comparisons of type systems and control flow systems by establishing correspondences between certain type systems and control flow analyses [7, 20]. Drawing inspiration from these, we develop a type and effect system which conservatively estimates the set of functions and channels which may be transmitted by an expression in our Facile-like language Mobile-λ.

Section 2 motivates the problem by an informal discussion and examples. Section 3 introduces the language Mobile-λ on which our discussion is based. The type and effect system is defined in Section 4 and proved consistent with the dynamic semantics of Mobile-λ in Section 5. The analysis which estimates potentially mobile functions is presented in Section 6. The following sections discuss the contributions of our work by referring to related work in the literature along with possible future directions.

2 Potential Mobility

In this section we define informally what is meant by *potential mobility* of functions and channels throughout the paper. We follow closely the criteria determined by Knabe to guide programmers in identifying potentially mobile functions in his implementation of Facile. The examples are also inspired by the discussion in his dissertation [11]. In the following section we will define Mobile-λ a formal language which is suggestive of an intermediate language for Facile-like languages. For the purposes of this section, it is sufficient to note that the language has operators ! and ? corresponding to sending and receiving a value and recall from Section 1 the data management policy of Facile.

2.1 Mobile Functions

It should be relatively clear that any function which is passed directly as an argument to the send operator (!) is potentially mobile. Whenever a function f moves, it takes with it the functions contained in its closure. Therefore, we can argue that a function which is referred to by a mobile function and which is not defined within it is also mobile. The functions which are defined within function f are mobile only by virtue of being a part of its code. Detecting the mobility of f implies the mobility of functions nested within its body. Therefore, we need not consider these as mobile functions individually.

If the above criteria were sufficient, all the information needed for automatic detection of mobile functions would be available in the definition environment component of the function closure which contains the bindings of free identifiers referred to by the code. However, in the presence of higher-order functions, one needs to go beyond the information provided by the function closure. The following examples illustrate this point.

Example 1

Let us consider a function f defined as follows.

```
fun f h =  let fun g x = ... h x ...
           in
             chan ! g
           end
```

Obviously, g is potentially mobile. We can also deduce that h stands for a function referred to by g. The difficulty arises from the fact that h is a bound variable. In such a situation, all the functions which h can be bound to, in other words all the functions which f can be applied to, are potentially mobile. Detecting these functions is not straightforward since one needs to consider cases such as the following.

```
fun k x = ... f a ...;      (* a will be mobile *)
fun k x = ... f x ...;      (* any possible binding of x will be mobile *)
fun k x = ... x a ... ;
... k f ...;                (* a will be mobile *)
fun k h y = ... h y ...;
... k f ...;                (* any possible binding of y will be mobile *)
```

Example 2

A similar difficulty arises when a higher-order function is transmitted. Let us consider an expression which transmits a function f defined as follows.

```
let
  fun f x =  let fun g y = ... x ... y ...
             in g
             end
in
  chan ! f
end
```

The function f is potentially mobile and the function g is defined within f. According to our discussion at the beginning of this section, there is no immediate reason for g to be taken as potentially mobile. However, it should be noted that the function g escapes the definition of f because it is returned as a result. It may be the case that the computation arrives at a point where it is transmitted by some other code. Therefore, it appears to be a safer option to consider functions such as g as potentially mobile.

2.2 Mobile Channels

As we have noted in Section 1 data structures implementing channels in Facile reside on the node where they were created and do not move. By channel mobility we mean the extrusion of the scope of a channel by sending its name. When a channel c is sent along a channel c', the scope of c expands to include the expression receiving it.

The arguments for mobile functions above apply to mobile channels as well and our criteria for identifying mobile channels is the same as those for functions. This is easily justified as channels are also first-class values and all first-class values are treated uniformly in Facile-like languages.

3 The Mobile-λ Language

Our view of a system is a coupling of two sites each of which evaluates a single expression. Each expression possibly communicates with the expression on the remote site through synchronous channels. We believe that this simple setting embodies the core ingredients of a distributed system where the basic computational units are functions. It provides a sufficient level of generality to model the remote transmission of functions which is our main interest.

We introduce the Mobile-λ language which includes all the constructs of a typical sequential functional language extended with constructs for creating channels and sending and receiving values over these channels.

3.1 Syntax

The unusual point about the syntax is that function abstractions, recursive expressions and channel allocation expressions are labelled. The purpose of labels (l) drawn from a set of labels (L) is to uniquely identify functions and channels. Therefore, all labels in a program are required to be distinct.

$$
\begin{array}{lcl}
e & ::= & () \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{chan}^l() \\
 & & \mid x \mid \texttt{fn}^l\, x \Rightarrow e \mid e_1\, e_2 \mid \texttt{primop}\, c\, e \\
 & & \mid e_1; e_2 \mid \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \\
 & & \mid \texttt{rec}^l\, f(x) \Rightarrow e \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \\
 & & \mid e_1? \mid e_1!e_2
\end{array}
$$

3.2 Dynamic Semantics

3.2.1 Semantic Objects

Values of the language consist of primitive values, channel identifiers and function closures. We assume that the dynamic representation of a channel identifier encodes the label l of its creation point in the program and the site s where the channel is created. The same applies to function closures, however, function closures also include the definition environment E of the function, its formal parameter x and its body e. It is possible for a channel allocation expression at a particular program point labeled with l to be evaluated several times. As an example, one could consider a channel allocation expression occurring within a recursive function. In this case each evaluation of the expression should generate a new channel identifier. We address this by tagging s with a counter n, where n is a freshly generated number for each new channel. We also use semantic objects which describe the communication behaviour of an expression; com stands for a communication event drawn from the set $\{\mathsf{new}\, ci, ci\,?\,val, ci!val\}$ representing the allocation of a channel, receiving and sending values over a channel respectively. The name w stands for a sequence of communication events. Note that a purely sequential computation which does not involve any communication is represented by ϵ.

An evaluation environment E is defined as a finite map from identifiers to values.

$$
\begin{array}{lcl}
ci & ::= & l@(n,s) \\
closure & ::= & \langle l, E, x, e\rangle@s \\
val & ::= & val_{unit} \mid val_{true} \mid val_{false} \mid val_n \\
 & & ci \mid closure \\
com & ::= & \mathsf{new}\, ci \mid ci?val \mid ci!val \\
w & ::= & \epsilon \mid w.com
\end{array}
$$

3.2.2 Top-level Rule and Matching

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. Figure 1 shows the evaluation of a system in which expression e_1 is evaluated at site s_1 and expression e_2 is evaluated at site s_2. We represent the top-level environment as a pair of environments and the result produced by the system as a pair of values. The top-level evaluation decomposes into individual evaluations at each site. The dynamic semantics rules for evaluation of an expression are further explained in Section 3.2.3.

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

Figure 1: Concurrent Evaluation

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. Informally speaking, at the end of the evaluation, we can observe that new channels have been allocated. Particular values transmitted on these channels will have been used in producing the resulting value, however, we treat their transmissions as internal actions of the system and abstract away from them.

Definition 1 ($\mathsf{IOmatch}(w_1, w_2)$)

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

3.2.3 Sequential Evaluation Rules

A judgement of the form $E \vdash_s e \overset{w}{\Longrightarrow} val$ states that expression e evaluates to value val against an environment E at site s. The evaluation relation

(1)	$\epsilon \parallel \epsilon \hookrightarrow \epsilon$
(2)	$\dfrac{w_1 \parallel w_2 \hookrightarrow w}{w_1.com \parallel w_2 \hookrightarrow w.com}$
(3)	$\dfrac{w_1 \parallel w_2 \hookrightarrow w}{w_1 \parallel w_2.com \hookrightarrow w.com}$
(4)	$\dfrac{w_1 \parallel w_2 \hookrightarrow w}{w_1.ci!val \parallel w_2.ci?val \hookrightarrow w}$
(5)	$\dfrac{w_1 \parallel w_2 \hookrightarrow w}{w_1.ci?val \parallel w_2.ci!val \hookrightarrow w}$

Figure 2: Event Matching

$\Longrightarrow$, which is defined in Figure 3, is annotated with the possible communication events which may occur during evaluation. 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 **(primop)** serves as a semantic tool to define the application of a predefined operation to an argument.

4 Type System

In this section we design a polymorphic type and effect system for Mobile-λ. Our aim is to estimate the functions and channels a Mobile-λ expression can possibly transmit by analysing the types and effects associated with it. We make use of the labels provided in the syntax to trace the flow of values through computation. The essential idea is to incorporate these labels into the types and effects so that we can eventually extract those which are of interest to us.

4.1 Semantic Objects

Our type system views types (τ) as a pair consisting of two components: a raw type ($\bar{\tau}$) and a mobility annotation (μ). Raw types classify values in the the conventional sense whereas mobility annotations estimate the identities of

(unit) $E \vdash_s () \xRightarrow{\epsilon} val_{unit}$

(true) $E \vdash_s \texttt{true} \xRightarrow{\epsilon} val_{true}$

(false) $E \vdash_s \texttt{false} \xRightarrow{\epsilon} val_{false}$

(int) $E \vdash_s n \xRightarrow{\epsilon} val_n$

(chan) $E \vdash_s \texttt{chan}^l() \xRightarrow{\mathsf{new}\,ci} ci \quad \text{where}\, ci = l@(n, s)$

(var) $E \vdash_s x \xRightarrow{\epsilon} E(x)$

(fn) $E \vdash_s \texttt{fn}^l\, x \ \Rightarrow\ e \xRightarrow{\epsilon} \langle l, E, x, e\rangle @s$

(rec)
$$\frac{val = \langle l, E[f \ \mapsto\ val],\ x,\ e\rangle @s}{E \vdash_s \texttt{rec}^l\, f(x) \Rightarrow e \xRightarrow{\epsilon} val}$$

(app)
$$\frac{E \vdash_s e_1 \xRightarrow{w_1} \langle l', E',\ x,\ e\rangle @s' \quad E \vdash_s e_2 \xRightarrow{w_2} val \quad E'[x \ \mapsto\ val] \vdash_s e \xRightarrow{w_3} val'}{E \vdash_s e_1\, e_2 \xRightarrow{w_1.w_2.w_3} val'}$$

(primop)
$$\frac{E \vdash_s \texttt{primop}\, c \xRightarrow{w_1} b \quad E \vdash_s e \xRightarrow{w_2} val \quad \mathrm{APPLY}(b, val) = val_1}{E \vdash_s \texttt{primop}\, c\, e \xRightarrow{w} val_1}$$

(seq)
$$\frac{E \vdash_s e_1 \xRightarrow{w_1} val_1 \quad E \vdash_s e_2 \xRightarrow{w_2} val_2}{E \vdash_s e_1\, ;\, e_2 \xRightarrow{w_1.w_2} val_2}$$

(if-t)
$$\frac{E \vdash_s e_1 \xRightarrow{w_1} val_{true} \quad E \vdash_s e_2 \xRightarrow{w_2} val}{E \vdash_s \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \xRightarrow{w_1.w_2} val}$$

(if-f)
$$\frac{E \vdash_s e_1 \xRightarrow{w_1} val_{false} \quad E \vdash_s e_3 \xRightarrow{w_2} val}{E \vdash_s \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \xRightarrow{w_1.w_2} val}$$

(let)
$$\frac{E \vdash_s e_1 \xRightarrow{w_1} val_1 \quad E[x \ \mapsto\ val_1] \vdash_s e_2 \xRightarrow{w_2} val_2}{E \vdash_s \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \xRightarrow{w_1.w_2} val_2}$$

(receive)
$$\frac{E \vdash_s e \xRightarrow{w} ci}{E \vdash_s e? \xRightarrow{w.ci?val} val}$$

(send)
$$\frac{E \vdash_s e_1 \xRightarrow{w_1} ci \quad E \vdash_s e_2 \xRightarrow{w_2} val}{E \vdash_s e_1 ! e_2 \xRightarrow{w_1.w_2.ci!val} val_{unit}}$$

Figure 3: Sequential Evaluation Rules

values which become mobile upon the transmission of the value of that raw type. We distinguish mobility annotations for channels (χ) and functions (Φ). We also have 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.

$$
\begin{array}{lll}
\bar{\tau} & ::= & \mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{chan}(\bar{\tau}) \mid \tau \xrightarrow{\kappa} \tau \mid \alpha \\
\tau & ::= & (\mathsf{Unit}, \emptyset) \mid (\mathsf{Int}, \emptyset) \mid (\mathsf{Bool}, \emptyset) \mid (\mathsf{chan}(\bar{\tau}), \chi) \mid (\tau \xrightarrow{\kappa} \tau, \Phi) \\
\Phi & ::= & \langle \varepsilon, l \rangle \mid \gamma \mid \Phi \cup \Phi \\
\chi & ::= & l \mid \rho \mid \chi \cup \chi \\
\kappa & ::= & \emptyset \mid \mathsf{new}(\chi, \bar{\tau}) \mid \mathsf{send}(\chi, \tau) \mid \mathsf{recv}(\chi, \bar{\tau}) \mid \beta \mid \kappa \cup \kappa \\
\mu & ::= & \emptyset \mid \chi \mid \Phi
\end{array}
$$

Mobility annotations are represented by sets. We associate the empty set with base types as we are not concerned with the mobility of values of base types. For example, if the raw type indicates that the expression is of type Int, this information is sufficient. We need not estimate which particular integers its value can be. However, as we are interested in the mobility of channels and functions we associate them with non-empty annotations.

The simplest form of mobility annotation for channels is l. An expression which receives a mobility annotation of the form l statically would evaluate at run-time to the channel which is allocated by the expression labelled with l. It is not always possible to statically infer such precise information. The approximate set of channels an expression can evaluate to is expressed by the union of simple channel annotations. The variable ρ ranges over mobility annotations for channels.

The simplest form of mobility annotation for functions is $\langle \varepsilon, l \rangle$. The operation of forming such an annotation is formally defined by Definition 4. The idea behind this definition is that we narrow down the static environment to the bindings of free identifiers of the function body, call it ε, and encapsulate it in the mobility annotation of the function along with the label l of the function. As was the case for channels, we allow mobility annoations to be a union of simple annotations. The variable γ ranges over mobility annotations for functions.

The set κ estimates the communication effects which may occur during the evaluation of an expression. An expression is associated with $\emptyset$ if its evaluation incurs no communication, whereas it is associated with the effects $\mathsf{new}(\chi, \bar{\tau}), \mathsf{send}(\chi, \tau), \mathsf{recv}(\chi, \bar{\tau})$ if its evaluation may incur channel allocation, send and receive events respectively. The effects can be merged by the set union as was the case for mobility annotations. The variable β ranges over effects.

By parameterising channels over raw types ($\bar{\tau}$) we allow the expression of channels which may communicate values of the same raw type regardless of their mobility annotations. Parameterising channels over (τ) would lead to restrictions in expressiveness as a channel would be allowed to communicate a certain set of values only. The channel allocation and receive effects are also parameterised over raw types. However, we parameterise the send effects over types with mobility annotations, because these effects are what we will ultimately analyse to detect the potentially mobile values.

We define the range of a communication effect $Rng(\kappa)$ as the set of pairs (χ, τ) or $(\chi, \bar{\tau})$ such that either $\mathsf{new}(\chi, \bar{\tau})$, $\mathsf{recv}(\chi, \bar{\tau})$ or $\mathsf{send}(\chi, \tau)$ is in κ.

4.1.1 Variables and Substitutions

We have included variables $\alpha, \beta, \gamma, \rho$ as static semantic objects to be able to express polymorphism of functions. We write $fv(\tau)$ for the set of free raw type, mobility annotation and effect variables in τ. The function fv extends pointwise to environments ε in the expected way.

$$
\begin{array}{ll}
fv(\bar{\tau}, \mu) = fv(\bar{\tau}) \cup fv(\mu) & fv(\langle \varepsilon, l \rangle) = fv(\varepsilon) \cup fv(l) \\
fv(\mathsf{Unit}) = \emptyset & fv(\gamma) = \{\gamma\} \\
fv(\mathsf{Int}) = \emptyset & fv(\Phi_1 \cup \Phi_2) = fv(\Phi_1) \cup fv(\Phi_2) \\
fv(\mathsf{Bool}) = \emptyset & fv(l) = \emptyset \\
fv(\mathsf{chan}(\bar{\tau})) = fv(\bar{\tau}) & fv(\rho) = \{\rho\} \\
fv(\tau_1 \xrightarrow{\kappa} \tau_2) = fv(\tau_1) \cup fv(\tau_2) \cup fv(\kappa) & fv(\chi_1 \cup \chi_2) = fv(\chi_1) \cup fv(\chi_2) \\
fv(\alpha) = \{\alpha\} &
\end{array}
$$

$$
\begin{array}{l}
fv(\mathsf{new}(\chi, \bar{\tau})) = fv(\chi) \cup fv(\bar{\tau}) \\
fv(\mathsf{recv}(\chi, \bar{\tau})) = fv(\chi) \cup fv(\bar{\tau}) \\
fv(\mathsf{send}(\chi, \tau)) = fv(\chi) \cup fv(\tau) \\
fv(\beta) = \{\beta\} \\
fv(\kappa_1 \cup \kappa_2) = fv(\kappa_1) \cup fv(\kappa_2)
\end{array}
$$

Vector notation is used to represent sequences of variables, for example $\vec{\alpha}, \vec{\beta}, \vec{\gamma}$. Throughout the paper we refer to mobility annotation variables collectively as δ and if it is not necessary to specify what a variable represents we will use ι.

Substitutions θ are finite maps from raw type variables α to raw types $\bar{\tau}$, effect variables β to effects κ and mobility annotations variables γ, ρ, δ to mobility annotations Φ, χ, μ.

4.1.2 Type schemes and Environments

Type schemes and environments are familiar from polymorphic type systems for languages with a functional core.

Definition 2 (Type Scheme (σ))

Type schemes (σ) are defined as types quantified over a set of raw type, effect and mobility annotation variables.

$$\sigma ::= \forall \vec{\alpha}\vec{\beta}\vec{\delta}\vec{\gamma}\vec{\rho}.\tau$$

The context in which an expression is associated with a raw type, an effect and a mobility annotation is represented by a static environment Γ, which maps value identifiers to type schemes (σ). The definition of fv above extends to type schemes by $fv(\forall \vec{\iota}.\tau) = fv(\tau) \setminus \vec{\iota}$, to type environments ($\Gamma$) and narrowed down type environments (ε). The domain of an environment Γ is referred to as $Dom(\Gamma)$. A variable is free in Γ if and only if it is free in $\Gamma(x)$ for some x in $Dom(\Gamma)$. The definition of a free variable in ε follows similarly. The notation $\Gamma[x \mapsto \sigma]$ is used for adding element x to the environment Γ, overriding the existing binding if x is already in the domain of Γ.

Definition 3 (Type Generalisation)

A type scheme $\sigma = \forall \vec{\alpha}\vec{\beta}\vec{\delta}\vec{\gamma}\vec{\rho}.\tau$ generalises a type τ', written as $\sigma \succ \tau'$, if there exists a substitution θ over the bound variables of τ such that $\theta\tau = \tau'$. Types are generalised to type schemes by the operation *Gen*. A variable cannot be generalised if it is free in the environment Γ or effect κ.

$Gen(\kappa, \Gamma)(\tau) = \mathtt{let}\{\vec{\alpha}, \vec{\beta}, \vec{\delta}, \vec{\gamma}, \vec{\rho}\} = fv(\tau) \setminus (fv(\kappa) \cup fv(\Gamma))\ \mathtt{in}\ \forall \vec{\alpha}\vec{\beta}\vec{\delta}\vec{\gamma}\vec{\rho}.\,\tau.$

Definition 4 (Environment Narrowing)

The operation *Narrow* narrows down a static environment Γ with respect to an expression. $Narrow(\Gamma, e, S) = \varepsilon$ where ε is the environment Γ restricted to the bindings of free identifiers of e with function or channel types, excluding the identifiers in set S.

4.2 Typing Rules

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 4. We comment on some of the rules below pointing at the characteristic features of our type system.

The rule **(chan)** shows that a channel allocation expression labelled with l assigns a mobility annotation l to the expression. The type of the values which will be communicated on this channel is not known at this stage, hence the variable type. The presence of this variable type in the associated effect is instrumental in performing type generalisation in a safe way.

Regarding polymorphism, we adopt the standard discipline employed in languages with a functional core and allow type generalisation to be performed in the **(let)** rule only.

The rule **(if)** shows the merging of mobility annotations. It is not always possible to statically detect which branch of the `if` statement will be taken at run-time. Merging of mobility annotations leads to the conservative estimation of the value of the `if` expression in question.

The rule **(receive)** is applied in typing a `receive` expression. As a `receive` expression evaluates to the value it receives, its type is expected to be the same as that of the received value. In our type system channels are parameterised over raw types. Therefore, the mobility annotation of a `receive` expression cannot be inferred from the type of the channel it listens to. This is not a significant deficiency, however, as the send effects are parameterised over mobility annotated types. The information we seem to have lost in the type of a `receive` expression is available in the type of the corresponding `send` expression.

The rule **(send)** is essential for our type system. The type assigned to a send expression is always Unit whereas the effect embodies the type of the transmitted value. The fact that a send effect is parameterised over a mobility annotated type makes it possible to analyse these effects further to expose potentially mobile values. We address this in Section 6.

The subsumption rule **(subs)** allows an expression to admit a larger communication effect than seems to be necessary. We adopt the approach which is referred to as early subsumption in [18] and subeffecting in [26]. This is in contrast to the approach known as late subsumption or subtyping where coercion can happen at any time inside any type. The advantage of our approach is that the complex interplay between polymorphism and subtyping can be avoided.

4.3 An Example

Having defined Mobile-$\lambda$and its type system it can now be instructive to present an example. We revisit Example 1 from Section 2 and consider its translated form in the syntax of Mobile-λ.

$$
\begin{array}{ll}
\texttt{let}\, f = \texttt{fn}^{l_1}\, h \;\Rightarrow & \texttt{let}\, g = \texttt{fn}^{l_2} x \;\Rightarrow\; \ldots\, h\; x\, \ldots \\
& \texttt{in}\;\; chan\,!\,g \\
& \texttt{end} \\
\texttt{in} & (*\text{ rest of the program } *)
\end{array}
$$

We expect the type of function f to express the following fact. Whenever f is applied to an argument function g will become mobile and since g refers

(unit)	$\Gamma \vdash_s () : (\mathsf{Unit}, \emptyset), \emptyset$
(true)	$\Gamma \vdash_s \mathtt{true} : (\mathsf{Bool}, \emptyset), \emptyset$
(false)	$\Gamma \vdash_s \mathtt{false} : (\mathsf{Bool}, \emptyset), \emptyset$
(int)	$\Gamma \vdash_s n : (\mathsf{Int}, \emptyset), \emptyset$
(chan)	$\Gamma \vdash_s \mathsf{chan}^l() : (\mathsf{chan}(\alpha), l), \mathsf{new}(l, \alpha) \quad \alpha \text{ new}$
(var)	$\dfrac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash_s x : \tau, \emptyset}$
(fn)	$\dfrac{\Gamma[x \mapsto \tau_1] \vdash_s e : \tau_2, \kappa \quad \varepsilon = \mathit{Narrow}(\Gamma, e, \{x\})}{\Gamma \vdash_s \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle), \emptyset}$
(rec)	$\dfrac{\Gamma[f \mapsto (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle)][x \mapsto \tau_1] \vdash_s e : \tau_2, \kappa \quad \varepsilon = \mathit{Narrow}(\Gamma, e, \{x\})}{\Gamma \vdash_s \mathtt{rec}^l\, f\,(x) \Rightarrow e : (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle), \emptyset}$
(app)	$\dfrac{\Gamma \vdash_s e_1 : (\tau_1 \xrightarrow{\kappa} \tau_2, \phi), \kappa' \quad \Gamma \vdash_s e_2 : \tau_1, \kappa''}{\Gamma \vdash_s e_1\, e_2 : \tau_2, \kappa \cup \kappa' \cup \kappa''}$
(primop)	$\dfrac{\Gamma \vdash_s \mathtt{primop}\, c : (\tau_1 \xrightarrow{\kappa} \tau_2, \emptyset), \kappa \quad \Gamma \vdash_s e : \tau_1, \kappa'}{\Gamma \vdash_s \mathtt{primop}\, c\, e : \tau_2, \kappa \cup \kappa'}$
(seq)	$\dfrac{\Gamma \vdash_s e_1 : \tau_1, \kappa \quad \Gamma \vdash_s e_2 : \tau_2, \kappa'}{\Gamma \vdash_s e_1 ; e_2 : \tau_2, \kappa \cup \kappa'}$
(if)	$\dfrac{\Gamma \vdash_s e_1 : (\mathsf{Bool}, \emptyset), \kappa, \quad \Gamma \vdash_s e_2 : (\bar{\tau}, \mu), \kappa' \quad \Gamma \vdash_s e_3 : (\bar{\tau}, \mu'), \kappa''}{\Gamma \vdash_s \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : (\bar{\tau}, \mu \cup \mu'), \kappa \cup \kappa' \cup \kappa''}$
(let)	$\dfrac{\Gamma \vdash_s e_1 : \tau_1, \kappa \quad \Gamma[x \mapsto \mathit{Gen}(\kappa, \Gamma)(\tau_1)] \vdash_s e_2 : \tau_2, \kappa'}{\Gamma \vdash_s \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \tau_2, \kappa \cup \kappa'}$
(receive)	$\dfrac{\Gamma \vdash_s e : (\mathsf{chan}(\bar{\tau}), \chi), \kappa \quad \delta \text{ new}}{\Gamma \vdash_s e? : (\bar{\tau}, \delta), \kappa \cup \mathsf{recv}(\chi, \bar{\tau})}$
(send)	$\dfrac{\Gamma \vdash_s e_1 : (\mathsf{chan}(\bar{\tau}), \chi), \kappa \quad \Gamma \vdash_s e_2 : (\bar{\tau}, m), \kappa'}{\Gamma \vdash_s e_1 ! e_2 : \mathsf{Unit}, \kappa \cup \kappa' \cup \mathsf{send}(\chi, (\bar{\tau}, \mu))}$
(subs)	$\dfrac{\Gamma \vdash_s e : \tau, \kappa \quad \kappa \subseteq \kappa'}{\Gamma \vdash_s e : \tau, \kappa'}$

Figure 4: Typing Rules

to the argument of f (the binding of h), the argument of f will also be mobile.

Let us consider a typing context as follows where we refer to the types of h and g as type(h)and type(g) for convenience.

$$
\begin{aligned}
&x : (\mathsf{Int}, \emptyset) \\
&h : ((\mathsf{Int}, \emptyset) \xrightarrow{\beta} (\mathsf{Int}, \emptyset), \gamma) \\
&g : ((\mathsf{Int}, \emptyset) \xrightarrow{\beta'} (\mathsf{Bool}, \emptyset), \langle [h \mapsto \mathrm{type}(h)], l_2 \rangle)
\end{aligned}
$$

Note that we have further assumed that h is the only free identifier in function g. Since g must have been typed in the context where h occurs free, due to Definition 3, β and γ in the type of h have not been generalised. We can now infer the type of f to be as follows.

$$
f : \forall \beta \forall \gamma. (((\mathsf{Int}, \emptyset) \xrightarrow{\beta} (\mathsf{Int}, \emptyset), \gamma) \xrightarrow{\kappa_f} (\mathsf{Unit}, \emptyset), \langle \varepsilon_f, l_1 \rangle)
$$

where $\kappa_f = \mathsf{send}(l_{chan}, \mathrm{type}(g))$.

Since the type of h does not appear free in the context in which f is typed, the operation *Gen* from Definition 3 would generalise β and γ in this case. The typing rules also impose that $chan \in Dom(\varepsilon_f)$. The generalisation of β and γ illustrate the expressive power of the type system. The function f can be applied uniformly to any function from integers to integers.

We could now convince the reader that the type of f includes all the information we seek. Whenever f is applied, its polymorphic type will be specialised to that of its argument, causing the generalised mobility annotation of h to be specialised to the mobility annotation of its argument. Since the type of h also appears in the latent effect κ_f, through being present in type(g), the effect associated with the application will reveal the type of h. For example, if at some point in the rest of the program, there appears an expression $f\ k$ where

$$
k : ((\mathsf{Int}, \emptyset) \xrightarrow{\kappa} (\mathsf{Int}, \emptyset), \langle [], l_k \rangle)
$$

the effect associated with $f\ k$ will be $\mathsf{send}(l_{chan}, \mathrm{type}(g))$ where γ in type(g) will specialise to $(\langle [], l_k \rangle)$. In Section 6 we present an analysis which shows how to extract the mobility annotations from the types systematically, such as extracting l_k from the type of g in this example.

5 Semantic Consistency

We define a consistency judgement of the form $w : \kappa, CE \models val : (\bar{\tau}, \mu)$ which relates objects of the dynamic semantics to objects of the static semantics with respect to a given channel environment CE where CE is defined as a map from channel identifiers to channel annotation and raw type pairs:

$$CE ::= [ci_1 \mapsto (\chi_1, \bar{\tau}_1), \ldots, ci_n \mapsto (\chi_n, \bar{\tau}_n)]$$

The intuition behind our consistency judgement is as follows. In Mobile-$\lambda$the changes in the state occur due to the communication events which are represented by w in the dynamic semantics. The communication effects κ are the static counterparts of these events. Thus, the $w : \kappa$ component of the judgement expresses the consistency of dynamic and static representations of state changes. The channel environment CE, relates channel identifiers to their mobility annotations and the raw type of the values they communicate. By fixing the raw type of the values which a channel is allowed to communicate, we preclude the loopholes for breaking the type system. Just as we require the dynamic representation of state changes to be consistent with the static ones, we require the values produced by evaluation of expressions to be consistent with the types assigned to them statically. This is expressed by the $val : (\bar{\tau}, \mu)$ part of the judgement.

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

$w : \kappa, CE \models val_{unit} : (\mathsf{Unit}, \emptyset)$
$w : \kappa, CE \models val_{true} : (\mathsf{Bool}, \emptyset)$
$w : \kappa, CE \models val_{false} : (\mathsf{Bool}, \emptyset)$
$w : \kappa, CE \models val_{n} : (\mathsf{Int}, \emptyset)$

$w : \kappa, CE \models ci : (\mathsf{chan}(\bar{\tau}), \chi)$ where $ci = l@(n, s)$
if $(l, \bar{\tau}) \in Rng(\kappa), CE(ci) = (\chi, \bar{\tau}), l \subseteq \chi$,
and for any val such that $ci\,?\,val \in w$ or $ci!val \in w$
$w : \kappa, CE \models val : (\bar{\tau}, \mu)$ for some μ
$w : \kappa, CE \models closure : (\tau_1 \xrightarrow{\kappa'} \tau_2, \Phi)$ where $closure = \langle l, E, x, e\rangle@s$
if there exists a Γ such that $\Gamma \vdash_s \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa'} \tau_2, \langle \varepsilon, l\rangle), \emptyset$,
$\langle \varepsilon, l\rangle \subseteq \Phi$ and $w : \kappa, CE \models E : \Gamma$

We write $w : \kappa, CE \models val : \forall \vec{\iota}.\tau$ 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 $\mathsf{Dom}(E) = \mathsf{Dom}(\Gamma)$,
  and $w : \kappa, CE \models E(x) : \Gamma(x)$ for any $x \in \mathsf{Dom}(E)$

The Definitions 5 and 6 above do not define the consistency relation in an inductive fashion as the cases for channels and functions refer to the consistency of semantic objects which we cannot recover from the structure of the channel or the function value. In [22, 25] Tofte, Talpin and Jouvelot discuss in detail why Definitions such as 5 and 6 should be regarded as the property of a consistency relation rather than the relation itself. We define the consistency relation as the maximal fixed point of the property stated in Definition 5 as is typically the method for co-inductive definitions. We consider below the appropriate functional F whose maximal fixed point gives us the consistency relation.

Definition 7 (F)

Let R be the set of all tuples $(w, \kappa, CE, val, (\bar{\tau}, \mu))$. The function F is defined over all elements Q of the powerset $P(R)$ of R. The greatest fixed point of F, $Q^{max} = \bigcup\{Q \subseteq R \mid Q \subseteq F(Q)\}$ satisfies the property stated in Definition 5. We say that $w : \kappa, CE \models val : (\bar{\tau}, \mu)$ if and only if $(w, \kappa, CE, val, (\bar{\tau}, \mu)) \in Q^{max}$.

$$
\begin{array}{rcl}
F(Q) & = & \{(w, \kappa, CE, val, (\bar{\tau}, \mu)) \mid \\
& & \text{if } val = val_{unit} \text{ then } \bar{\tau} = \mathsf{Unit} \text{ and } v = \emptyset \\
& & \text{if } val = val_{true} \text{ then } \bar{\tau} = \mathsf{Bool} \text{ and } v = \emptyset \\
& & \text{if } val = val_{false} \text{ then } \bar{\tau} = \mathsf{Bool} \text{ and } v = \emptyset \\
& & \text{if } val = val_n \text{ then } \bar{\tau} = \mathsf{Int} \text{ and } v = \emptyset \\
& & \\
& & \text{if } val = l@(n, s) \text{ then there exists a } \bar{\tau}' \text{ such that} \\
& & \bar{\tau} = \mathsf{chan}(\bar{\tau}'), (l, \bar{\tau}') \in Rng(\kappa), CE(ci) = (\mu, \bar{\tau}'), \\
& & l \subseteq \mu \text{ and } (w, \kappa, CE, val', \tau') \text{ for any } val' \text{ such that} \\
& & ci?val' \in w \text{ or } ci!val' \in w \\
& & \\
& & \text{if } val = \langle l, E, x, e\rangle@s \text{ then there exists a } \Gamma \text{ such that} \\
& & \Gamma \vdash_s \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa'} \tau_2, \langle \varepsilon,\, l\rangle), \emptyset, \text{and } \langle \varepsilon,\, l\rangle \subseteq \Phi \\
& & \text{and } (w, \kappa, CE, E(x), \tau') \text{ for any } x \in Dom(E) \text{ and } \Gamma(x) \succ \tau'\}
\end{array}
$$

For Q^{max} to exist the above defined operator F must be monotonic, meaning if $Q \subseteq Q'$ then $Q \subseteq F(Q)$. It is straightforward to show this property. Therefore, we skip this part of the proof and assume that F is monotonic in the rest of the paper.

Definition 8 (Extension)

(κ', CE') extends (κ, CE), written $(\kappa, CE) \sqsubseteq (\kappa', CE')$ 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 9 (Equivalence)

(κ, CE) and (κ', CE') are equivalent, written $(\kappa, CE) \simeq (\kappa', CE')$, if and only if $(\kappa, CE) \sqsubseteq (\kappa', CE')$ and $(\kappa', CE') \sqsubseteq (\kappa, CE)$.

Definition 10 (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.

Proposition 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. The 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.

$$
\begin{array}{l}
E_1 \vdash_{s_1} e_1 \stackrel{w_1}{\Longrightarrow} val_1,\ E_2 \vdash_{s_2} e_2 \stackrel{w_2}{\Longrightarrow} val_2,\ \mathsf{IOmatch}(w_1, w_2), \\
\Gamma_1 \vdash_{s_1} e_1 : \tau_1, \kappa_1,\ \Gamma_2 \vdash_{s_2} e_2 : \tau_2, \kappa_2, \\
w_0 : \kappa_0, CE \models E_1 : \Gamma_1 \text{ and } w_0 : \kappa_0, CE \models E_2 : \Gamma_2
\end{array}
$$

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 $w_1 \mid\mid w_2 \hookrightarrow w$. $\square$

Appendix A contains the major lemmas used in the proof of the consistency. Selected cases from the proof of Proposition 1 are given in Appendix B. The method used in the proof of consistency depends on the approach used in the specification of the dynamic and static semantics. Relative merits of different methods in proving the two semantics consistent have been discussed in the related literature [12, 31, 22, 25]. Our formulation can be likened to that of [25] in the sense that we use the formalism of relational semantics in specifying the evaluation rules and the static semantics rules involve effect inference. It can also be likened to that of [12] in the sense that the formalism of relational semantics is enriched by labelling evaluation rules with event sequences to capture the communication behaviour.

6 Mobility Analysis

In the previous sections we defined the dynamic and static semantics and proved their consistency. In this section we continue in a similar fashion. We first define a function *Mobile* which examines the dynamic representation of a value *val* to extract the labels of those values which becomes mobile through the transmission of *val*. This function serves as a tool to formalise our notion of mobility of a value. We then give the specification of a static analysis which operates on the types of the static semantics to conservatively estimate the labels of potentially mobile channels and functions. The analysis is defined by means of an inference system which associates a type with two sets of labels, labels of potentially mobile channels and functions respectively. Finally, we state and prove the soundness of this analysis.

6.1 Mobile Values

Definition 11 (Mobile)

The function *Mobile* collects the labels of interest in the dynamic representation of a value in a set where the labels of channels are tagged with chan and the labels of functions are tagged with fn.

$$
\begin{array}{ll}
\mathit{Mobile}(\mathit{val}) = \emptyset & \text{for } \mathit{val} \in \{\mathit{val}_{\mathit{unit}}, \mathit{val}_{\mathit{true}}, \mathit{val}_{\mathit{false}}, \mathit{val}_n\} \\
\mathit{Mobile}(\mathit{val}) = \{(\mathsf{chan}, l)\} & \text{for } \mathit{val} = l@(n, s) \\
\mathit{Mobile}(\mathit{val}) = \{(\mathsf{fn}, l)\} \cup S & \text{for } \mathit{val} = \langle l, E, x, e\rangle@s \\
 & \text{where } S = \bigcup\{\mathit{Mobile}(E'(x')) \mid \\
 & \qquad\qquad E' = \mathit{Narrow}(E, e, x), \\
 & \qquad\qquad x' \in \mathit{Dom}(E'), \\
 & \qquad\qquad x' \neq \langle l, E, x, e\rangle@s\}.
\end{array}
$$

In the case of channels, the label of the channel identifier is extracted. In the case of a function closure, in addition to the label of the function itself, the environment part of the closure is examined to collect the labels bound to the free identifiers in the function body. We make use of the auxiliary function *Narrow* to narrow down a dynamic environment E to the bindings of free identifiers of the function body. The last condition in the definition of set the S ensures that the function *Mobile* does not enter an infinite loop while examining the environment of a recursively defined function.

6.2 Static Analysis

As we provided informal explanations on our particular choice of annotations in Section 4, at this point it should be relatively clear how we intend to use the static information captured in types to estimate mobile functions and channels. Our approach is as follows. Supposing that an expression is associated with an effect κ we consider each $\mathsf{send}(\chi, \tau) \in \kappa$ in isolation, analyse τ to collect the mobility annotations (labels) of interest and finally merge the results. Noting that this is what is required to obtain the overall set of labels, we suffice with presenting the analysis of τ.

Definition 12 ($\vdash \tau : C, F$)

Given a type $\tau = (\bar{\tau}, \mu)$, the static analysis infers two sets C, F written as ($\vdash \tau : C, F$) where C is a set of channel labels and F is a set of function labels. The rules of the inference system are given in Figure 5.

In Section 1 we listed some of the intended applications of our mobility analysis. In order to explain how the analysis presented here relates to this previous discussion we can consider the problem of estimating the cost of mobility. Suppose the implementation of Mobile-λ adopts the data management policy of Facile where the function closures are copied and channels are remotely referenced upon transmission. If a function f is assigned the effect $\mathsf{send}(\chi, \tau)$ and the analysis yields a set of channel labels C and function labels F ($\vdash \tau : C, F$) one can interpret the result as follows. If function f is transmitted, the program may have to create and maintain network references for the channels whose labels are present in the set C. Similarly, the program may have to copy the closures of functions whose labels are in the set F. This gives an idea of the potential size of the data to be copied between remote sites.

(1)	$\vdash (\bar{\tau}, \mu) : \emptyset, \emptyset \quad \text{if } \bar{\tau} \text{ is a base type or variable}$
(2)	$\vdash (\mathsf{chan}(\bar{\tau}), \rho) : \emptyset, \emptyset$
(3)	$\vdash (\mathsf{chan}(\bar{\tau}), l) : \{l\}, \emptyset$
(4)	$\dfrac{\vdash \tau_2 : C_2, F_2}{\vdash (\tau_1 \xrightarrow{\kappa} \tau_2, \gamma) : C_2, F_2}$
(5)	$\dfrac{\begin{array}{l} C_1 = \bigcup\{C \mid x \in \mathsf{Dom}(\varepsilon), \vdash \varepsilon(x) : C, F \text{ and } \varepsilon(x) \neq \langle \varepsilon, l \rangle\} \\ F_1 = \bigcup\{F \mid x \in \mathsf{Dom}(\varepsilon), \vdash \varepsilon(x) : C, F \text{ and } \varepsilon(x) \neq \langle \varepsilon, l \rangle\} \\ \vdash \tau_2 : C_2, F_2 \end{array}}{\vdash (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle) : C_1 \cup C_2, \{l\} \cup F_1 \cup F_2}$
(6)	$\dfrac{\vdash (\bar{\tau}, v_1) : C_1, F_1 \quad \vdash (\bar{\tau}, v_2) : C_2, F_2}{\vdash (\bar{\tau}, v_1 \cup v_2) : C_1 \cup C_2, F_1 \cup F_2}$

Figure 5: Mobility Analysis

Remarks on the Rules of the Analysis

The Rule 1 applies when the raw type is a base type or a variable. Our type system does not keep track of values of base types and a variable type does not contain any specific information that we need to note. Therefore, we take C and F to be empty.

The Rules 2 and 3 apply when the type is a channel type and the channel is estimated by a variable or a singleton. A variable includes no specific information, hence the empty C in Rule 2. If the mobility annotation is a specific label we put it in C. In cases where labels are present in the set χ, Rule 6 collects these one by one and merges them.

The Rules 4 and 5 apply when the type is a function type. If a function annotation is represented by a variable we have no information about the environment of the function or its label. However, the result type may include information about the escaping values, therefore it needs to be examined. Rule 4 handles this case. If the mobility annotation is a singleton, Rule 5 extracts the label of the function and examines the bindings of the identifiers of the narrowed down type environment to collect the labels of potentially mobile values. The return type is also examined to handle the escaping values. If mobility annotation is a non-singleton set Rule 6 ensures that each element of the set is examined and the results are merged into a single set.

Note also that infinite loops in examining the environments ε is prevented by a side condition in Rule 5.

6.3 Soundness

We have shown that our type system associates values with types which are consistent in the sense defined in Section 5. The soundness of the analysis presented above relies on this fact. The definition of the notion of soundness for the mobility analysis states that the analysis of a correctly typed expression would conservatively estimate the labels of interest. Conservative estimation is expressed by the subset relation.

Proposition 2 (Soundness of Analysis)

Suppose

$$\begin{array}{l} w : \kappa, CE \models val : \tau \\ MChan = \{l \mid (\mathsf{chan}, l) \in Mobile(val)\} \\ MFun = \{l \mid (\mathsf{fn}, l) \in Mobile(val)\} \\ \vdash \tau : C, F \end{array}$$

then

$$MChan \subseteq C \text{ and } MFun \subseteq F.$$

Proof (Soundness of Analysis) The proof is by induction on the derivation of $\vdash$ and selected cases can be found in Appendix C. □

7 Related Work

Initially, effect systems were proposed as a solution to the problem of safely integrating functional and imperative features [13]. The basic idea was to enhance the type systems so that the expressions were associated with their observable side effects as well as types. This approach was pursued by Talpin and Jouvelot to safely type references in ML-like languages [25] and by Tofte and Talpin to improve memory management techniques [30]. The same approach was later carried over to concurrent programming languages where effect information was used in safe generalisation of channel types. The work by Thomsen on the language Facile [28] and Bolignano and Debbabi on Concurrent ML [4] are examples of this.

Another line of research in the literature demonstrates that the exploitation of type and effect systems need not be confined to the enforcement of type safety in the above sense. Annotated with effects and other kinds of information, types can capture a significant amount of static information about a program's potential dynamic behaviour. For a more detailed explanation the reader is referred to the work by Nielson and Nielson which includes an analysis of communication topology of programs [18] and an analysis developed for obtaining information to be used in static and dynamic processor allocation [19].

Our work bears a strong similarity to the earlier type and effect systems, in particular to those designed for Facile and Concurrent ML. A common point of these systems is that they rely on the concept of channel regions which are the static counterparts of channels created at run-time. Channels play the key role in the problems they focus on. In our case however, it is not only channels but also functions which need to be traced. Therefore, we introduce a more general concept: mobility annotations. For channels, mobility annotations are analogous to regions.

8 Conclusions and Future Work

We have shown that the methodology of annotated type and effect systems can be exploited to provide a general profile of programs with respect to the potential mobility of values where the main computational units are functions. Such a profile could be put to use internally by a compiler to direct optimisations or could serve as a tool for programmers.

It is now becoming a common practice to use types in stipulating safety and security properties and to provide sound systems for reasoning about these properties. The type-based approach to security emphasises the significance of tracing the flow of information through computation [9]. The power of our type and effect system comes from its ability to trace the flow of values. In other words, the machinery to control secure information flow is already partially built into the type system. We can now look at ways of exploiting it for expressing, validating and enforcing certain safety and security properties.

Most of the work in the area of safety and security is in the framework of lower-level process calculi such as the π-calculus family and static analyses techniques developed for these [9, 24, 10, 32, 2, 3, 1]. Casting some of these in the framework of a type and effect system such as ours would make the underlying ideas more readily applicable using the type systems of languages such as Concurrent ML and Facile as a basis.

Acknowledgements

I would like to thank Stephen Gilmore for the very helpful discussions and the comments on the drafts of this paper. I also would like to thank Jane Hillston for her support. My research is funded by the University of Edinburgh.

References

[1] M. Abadi. Secrecy by typing in security protocols. In *Proceedings of Theoretical Aspects of Computer Software*, volume 1281 of *LNCS*, pages 611–638. Springer, 1997.

[2] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis for the π-calculus. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 1998.

[3] 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.

[4] D. Boloignano and M. Debbabi. A semantic theory for ML higher order concurrency primitives. In F. Nielson, editor, *ML with Concurrency*, chapter 6, pages 145–183. Springer-Verlag New York, Inc, 1997.

[5] K.L. Gasser, F.Nielson, and H.R.Nielson. Systematic realisation of control flow analysis for CML. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM Press, 1997.

[6] 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.

[7] N. Heintze. Control-flow analysis and type systems. Technical Report CMU-CS-94-227, School of Computer Science, Carnegie Mellon Univeristy, 1994. Also appears as Fox Memorandum CMU-CS-FOX-94-09.

[8] N. Heintze. Set-based analysis of ML programs. In *Proceedings of Lisp and Functional Programming*. ACM Press, 1994.

[9] 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.

[10] M. Hennesy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Proceedings of 4th Workshop on Mobile Object Systems*, volume 1603 of *LNCS*, pages 95–117, Brussels, 1998. Springer-Verlag.

[11] F.C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Dec. 1995.

[12] X. Leroy. *Polymorphic Typing of an Algorithmic Language*. PhD thesis, Univerity of Paris VII, 1992.

[13] J. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, New York, 1988. ACM.

[14] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[15] R. Milner, J.Parrow, and D. Walker. A calculus of mobile processes I and II. *Information and Computation*, 100:1–72, 1992.

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

[17] F. Nielson and H.R.Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*. ACM Press, 1997.

[18] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 84–97. ACM Press, 1994.

[19] H. R. Nielson and F. Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *Proceedings of TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 590–604. Springer-Verlag, 1995.

[20] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 367–378. ACM Press, 1995.

[21] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118:128–141, 1995.

[22] R.Milner and M.Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, Sep 1991.

[23] O. Shivers. *Control-Flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[24] G. Smith and D. Volpano. A type-based approach to program security. In *Proceedings of TAPSOFT '97*, pages 607–621, Lille, France.

[25] J.P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.

[26] Y. Tang and P. Jouvelot. Control-flow effects for escape analysis. In *Proceedings of WSA'92*, Bordeux, France, 1992.

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

[28] 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.

[29] B. Thomsen, L. Leth, S. Prasad, T. M. Kuo, A. Kramer, and F.C. Knabe. Facile Antigua Release programming guide. Technical Report ECRC-93-20, Europian Computer Industry Research Centre, Munich, Germany, Dec. 1993.

[30] M. Tofte and J.P. Talpin. Implemantation of the types call-by-value λ-calculus using a stack of regions. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 188–201, Portland, Oregon, USA, 1994.

[31] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical report, Rice University, 1992.

[32] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proceedings of International Conference on Concurrency Theory (CONCUR)*, LNCS. Springer-Verlag, 1999.

A Lemmas

The lemmas in this section are used in the proof of consistency selected cases of which are given in Appendix B. The proofs for lemmas 4, 6, 7, 8, 9 and 10 all follow the same technique as that of Lemma 2.

Lemma 1 (Substitution)

If $\Gamma \vdash e : \tau, \kappa$ then $\theta\Gamma \vdash e : \theta\tau, \theta\kappa$.

Proof (Substitution) The proof is given by induction on the derivation of $\Gamma \vdash e : \tau, \kappa$. Two selected cases are shown below.

$\boxed{\textbf{case } x}$ The rule **(var)** of the the static semantics imposes that the typing is of the following form $\Gamma \vdash_s x : \tau, \emptyset \;\; \Gamma(x) = \sigma$ and $\sigma \succ \tau$.

Suppose $\sigma = \forall \vec{\iota}.\tau_0$. After renaming if necessary we define θ so that $\vec{\iota}$ is out of reach of θ. Let θ' be a substitution over $\vec{\iota}$ such that $\theta'(\tau_0) = \tau$. Now define a substitution θ'' with domain $\vec{\iota_i}$ by $\theta''(\iota_i) = \theta(\theta''(\iota_i))$. We then have
$\theta''(\theta(\iota_i)) = \theta''(\iota_i) = \theta(\theta'(\iota_i))$ for all i
$\theta''(\theta(\beta)) = \theta(\beta) = \theta(\theta'(\beta))$ for all β not in $\vec{\iota_i}$
Hence, $\theta''(\theta(\tau_0)) = \theta(\theta'(\tau_0)) = \theta(\tau)$ which shows that $\theta(\tau)$ is an instance of $\theta(\Gamma(x))$. So, $\theta(\Gamma) \vdash_s x : \Gamma(\tau)$.

$\boxed{\textbf{case } \texttt{let}\, x = e_1 \,\texttt{in}\, e_2}$ The rule **(var)** of the the static semantics imposes that the typing is of the following form
$\Gamma \vdash_s e_1 : \tau_1, \kappa \quad \Gamma[x \;\mapsto\; Gen(\kappa, \Gamma)(\tau_1)] \vdash_s e_2 : \tau_2, \kappa'$

Let $\forall \vec{\iota}.\tau_1$ be $Gen(\kappa, \Gamma)(\tau_1)$. For any substitution θ, let us consider fresh $\vec{\iota'}$ and define θ' as the extension of $\theta_{\vec{\iota}}$ with $\theta\{\iota \mapsto \iota'\}$. By the definition of Gen we have that $\theta(\Gamma) = \theta'(\Gamma)$ and $\theta(\kappa) = \theta'(\kappa')$. By the definition of θ', ι being out of reach, $\theta'(\forall \vec{\iota}.\tau_1) = \forall \vec{\iota'}.\theta'\tau_1$. Thus,
$\theta(\Gamma[x \;\mapsto\; Gen(\kappa, \Gamma)(\tau_1)] = \theta(\Gamma)[x \mapsto Gen(\theta\kappa, \theta\Gamma)(\theta'\tau_1)]$.

By using the induction hypothesis on e_1 with θ' we get $\theta'\Gamma \vdash e : \theta'\tau, \theta'\kappa$. By definition of θ', $\theta'\Gamma \vdash e : \theta'\tau, \theta\kappa$. By using the induction hypothesis on e_2 we get $\theta(\Gamma[x \;\mapsto\; Gen(\kappa, \Gamma)(\tau_1)]) \vdash_s e_2 : \theta(\tau_2), \theta(\kappa')$ which is equivalent to $\theta(\Gamma)[x \;\mapsto\; Gen(\theta(\kappa), \theta(\Gamma))(\theta'(\tau_1))] \vdash_s e_2 : \theta(\tau_2), \theta(\kappa')$. Finally by definition of **(let)** we can conclude that $\theta(\Gamma) = \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 : \theta(\tau), \theta(\kappa)$. $\square$

Lemma 2 (Extension)

If $w : \kappa, CE \models val : \tau$ and $(\kappa, CE) \sqsubseteq (\kappa', CE')$ then $w : \kappa, CE' \models val : \tau$.

Proof (Extension) We consider the set:
$Z = \{(w, \kappa', CE', val, \tau) \mid w : \kappa, CE \models val : \tau \text{ and } (\kappa, CE) \sqsubseteq (\kappa', CE')\}$.
We show by case analysis on the structure of val that
$q = (w, \kappa', CE', val, \tau) \in F(Z)$. $\square$

Lemma 3 (Equivalence)

If $(\kappa, CE) \simeq (\kappa', CE')$ and $w : \kappa, CE \models val : \tau$ and if and only if $w : \kappa', CE' \models val : \tau$.

Proof (Equivalence) Since $(\kappa, CE) \simeq (\kappa', CE')$, by Definition 8, $(\kappa, CE) \sqsubseteq (\kappa', CE')$ and $(\kappa', CE') \sqsubseteq (\kappa, CE)$. If $w : \kappa, CE \models val : \tau$ then by Lemma 2, $w : \kappa, CE' \models val : \tau$. If $w : \kappa, CE' \models val : \tau$ then by Lemma 2, $w : \kappa, CE \models val : \tau$. $\square$

Lemma 4 (Semantic Substitution)

If $w : \kappa, CE \models val : \tau$ then $w : \theta\kappa, \theta CE \models val : \theta\tau$ for any substitution θ.

Lemma 5 (Instantiation)

If $w : \kappa, CE \models val : \tau$ and θ is defined on $fv(\tau) \setminus fv(\kappa)$ then $w : \kappa, CE \models val : \theta\tau$.

Lemma 6 (Channel allocation)

Let $\kappa' = \mathsf{new}(l, \bar{\tau})$, $w' = w.\mathsf{new}\, ci$ where $ci = l@(n, s)$ and $CE' = CE[ci \mapsto (l, \bar{\tau})]$ where $ci \notin Dom(CE)$.
If $w : \kappa, CE \models val : \tau'$ then $w' : \kappa \cup \kappa', CE' \models val : \tau'$.

Lemma 7 (Send)

Let $\kappa' = \mathsf{send}(\chi, \tau), \tau = (\bar{\tau}, \mu)$ for some μ,
$w' = w.ci!val, w : \kappa, CE \models ci : (\mathsf{chan}(\bar{\tau}), \chi)$ and $w : \kappa, CE \models val : \tau$.
If $w : \kappa, CE \models val' : \tau'$ then $w' : \kappa \cup \kappa', CE \models val' : \tau'$.

Lemma 8 (Communication)

If $w.ci!val : \kappa \cup \mathsf{send}(\chi, \tau)$ where $\tau = (\bar{\tau}, \mu)$ for some μ, $CE \models val : \tau$ and $\mathsf{IOmatch}(w, w')$ then $w'.ci?val : \kappa' \cup \mathsf{recv}(\chi, \bar{\tau}), CE \models val : \tau$ for some κ'.

Lemma 9 (Receive)

Let $\kappa' = \mathsf{recv}(\chi, \bar{\tau}), w' = w.ci?val, w : \kappa, CE \models ci : (\mathsf{chan}(\bar{\tau}), \chi)$ and $w' : \kappa \cup \kappa', CE \models val : (\bar{\tau}, \mu)$ for some μ. If $w : \kappa, CE \models val' : \tau'$ then $w' : \kappa \cup \kappa', CE \models val' : \tau'$.

Lemma 10 (Subsumption)

If $w : \kappa, CE \models val : \tau$ and $\kappa \subseteq \kappa'$ then $w : \kappa', CE \models val : \tau$.

B Consistency of the Dynamic and the Static Semantics

$\boxed{\textbf{Match Rule (1)} \quad \epsilon \mid\mid \epsilon \hookrightarrow \epsilon}$

If the evaluation respects match rule (1), the evaluation at both sites must have followed one of the dynamic semantics rules **(unit)**, **(int)**, **(true)**, **(false)**, **(var)**, **(fn)** and **(rec)**. We consider the evaluation of e_1 at site s_1 and show Part 1 of Proposition 1 for **(var)** and **(rec)**. The other cases are similar.

$\boxed{\textbf{case } e_1 = x}$ The rules **(var)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$E \vdash_{s_1} x \overset{\epsilon}{\Longrightarrow} E(x) \qquad \frac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash_{s_1} x : \tau, \emptyset}$$

By hypothesis $w_0 : \kappa_0, CE \models E : \Gamma$ which implies that for any $x \in \mathsf{Dom}(E)$

$$w_0 : \kappa, CE \models E(x) : \Gamma(x).$$

Since $\sigma \succ \tau$ where $\sigma = \Gamma(x)$, by Definition 5, $w_0 : \kappa, CE \models E(x) : \tau$. Now by taking $CE = CE', w_1 = \epsilon$ and $\kappa_1 = \emptyset$, we can derive the required conclusion $(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1, \kappa_0 \cup \kappa_1, CE')$ and $w_0.w_1 : \kappa_0 \cup \kappa_1, CE' \models E(x) : \tau$.

$\boxed{\textbf{case } e_1 = \mathtt{rec}^l\, f(x) \Rightarrow e}$ The rules **(rec)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$\frac{val = \langle l, E[f \mapsto val], x, e\rangle @ s_1}{E \vdash_{s_1} \mathtt{rec}^l\, f(x) \Rightarrow e \overset{\epsilon}{\Longrightarrow} val}$$

$$\frac{\Gamma[f \mapsto (\tau_1 \overset{\kappa}{\rightarrow} \tau_2, \langle \varepsilon, l\rangle)][x \mapsto \tau_1] \vdash_{s_1} e : \tau_2, \kappa \quad \varepsilon = \mathit{Narrow}(\Gamma, e, \{x\})}{\Gamma \vdash_{s_1} \mathtt{rec}^l\, f\,(x) \Rightarrow e : (\tau_1 \overset{\kappa}{\rightarrow} \tau_2, \langle \varepsilon, l\rangle), \emptyset}$$

To prove $w_0 : \kappa, CE \models val : (\tau_1 \overset{\kappa}{\rightarrow} \tau_2, \langle \varepsilon, l\rangle)$ where $\varepsilon = \mathit{Narrow}(\Gamma, e, \{x\})$ we define $Q = Q^{max} \cup \{(w_0, \kappa_0, CE, (val, \langle \varepsilon, l\rangle))\}$ and prove that $Q \subseteq F(Q)$. Take a $q \in Q$. If $q \in Q^{max}$ then $q \in F(Q)$ because $Q^{max} \subset Q$ and F is monotonic. Otherwise, $q = (w_0, \kappa_0, CE, (val, \langle \varepsilon, l\rangle))$. Let $\Gamma' = \Gamma[f \mapsto (\tau_1 \overset{\kappa}{\rightarrow} \tau_2, \langle \varepsilon, l\rangle)]$ and $E' = E[f \mapsto val]$. By the typing rule **(rec)** we have

$$\Gamma'[x \mapsto \tau_1] \vdash_{s_1} e : \tau_2, \kappa.$$

By the typing rule **(fn)** we can get

$$\Gamma' \vdash_{s_1} \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \overset{\kappa}{\rightarrow} \tau_2, \langle \varepsilon, l\rangle), \emptyset.$$

Since $w_0 : \kappa, CE \models E : \Gamma$, $\mathit{Dom}(E) = \mathit{Dom}(\Gamma)$ and for all $x \in \mathit{Dom}(E)$ we have $w_0 : \kappa, CE \models E(x) : \Gamma(x)$. Therefore, for all $x \in \mathit{Dom}(E)$, $(w_0, \kappa, CE, E(x), \Gamma(x)) \in Q$.

Moreover, $(w_0, \kappa_0, CE, (val, \langle \varepsilon, l\rangle)) \in Q$. Thus, $\mathit{Dom}(E') = \mathit{Dom}(\Gamma')$ and for all $x \in \mathit{Dom}(E')$ we have $(w_0, \kappa_0, CE, (E'(x), \Gamma(x)) \in Q$. By definition of F we have $q \in F(Q)$.

Match Rule (2) $\dfrac{w_1' \mid\mid w_2' \hookrightarrow w'}{w_1'.com \mid\mid w_2' \hookrightarrow w'.com}$

If the evaluation respects match rule (2), we need to consider all possible evaluations where the dynamic semantics rules **(chan), (app), (primop), (seq), (if)** and **(let)** may have been applied. Note that *com* can only be ϵ or $\mathsf{new}\,ci$ for some ci, as by hypothesis $\mathsf{IOmatch}(w_1, w_2)$ at the top-level where $w_1 = w_1'.com$ and $w_2 = w_2'$. We consider the evaluation of e_1 and show Part 1 of Proposition 1 for the cases where **(chan)** and **(let)** rules are applicable. The other cases are similar.

case $e_1 = \mathtt{let}\, x = e_1'\, \mathtt{in}\, e_2'$ The rules **(let)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$\frac{E \vdash_{s_1} e_1' \overset{w_1}{\Longrightarrow} val_1 \quad E[x \mapsto val_1] \vdash_{s_1} e_2' \overset{w_2}{\Longrightarrow} val_2}{E \vdash_{s_1} \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2' \overset{w_1.w_2}{\Longrightarrow} val_2}$$

$$\frac{\Gamma \vdash_{s_1} e_1' : \tau_1, \kappa \quad \Gamma[x \mapsto Gen(\kappa, \Gamma)(\tau_1)] \vdash_{s_1} e_2' : \tau_2, \kappa'}{\Gamma \vdash_{s_1} \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2' : \tau_2, \kappa \cup \kappa'}$$

Let us rename the variables $fv(\tau_1) \setminus fv(\kappa) \cup fv(\Gamma)$ avoiding capturing the names in $\kappa_0 \cup \kappa$ and Γ so that τ_1 becomes τ. By Lemma 1, $\Gamma \vdash_{s_1} e_1' : \tau, \kappa$. By induction hypothesis on e_1 there exists a CE' such that

$$(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1, \kappa_0 \cup \kappa, CE'),\ w_0.w_1 : \kappa_0 \cup \kappa, CE' \models val_1 : \tau.$$

Since $(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1, \kappa_0 \cup \kappa, CE')$, by Definition 5, $w_0.w_1 : \kappa_0 \cup \kappa, CE' \models E : \Gamma$. Let θ be any substitution defined on $\vec{\iota} = fv(\tau) \setminus fv(\kappa) \cup fv(\Gamma)$. Since $\vec{\iota} \setminus fv(\kappa) \cup fv(\Gamma) \cap fv(\kappa) = \emptyset$ and $\vec{\iota} \cap fv(\kappa) = \emptyset$, by Lemma 5,

$$w_0.w_1 : \kappa_0 \cup \kappa, CE' \models val_1 : \theta\tau.$$

By Definitions 3 and 5, $w_0.w_1 : \kappa_0 \cup \kappa, CE' \models val_1 : \vec{\iota}.\tau$. Take $E' = E[x \mapsto val_1]$ and $\Gamma' = \Gamma[x \mapsto \theta\tau]$. Now by induction hypothesis on e_2', we can get the sought result.

$\boxed{\textbf{case } e_1 = \mathtt{chan}^l()}$ The rules **(chan)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$E \vdash_{s_1} \mathtt{chan}^l() \overset{\mathsf{new}\, ci}{\Longrightarrow} ci \qquad \text{where } ci = l@(n, s)$$

$$\Gamma \vdash_{s_1} \mathtt{chan}^l() : (\mathsf{chan}(\alpha), l), \mathsf{new}(l, \alpha) \quad \alpha \text{ new}$$

Let us take $CE' = CE[ci \mapsto (l, \alpha)]$ such that α new and $ci \notin Dom(CE)$. Obviously, $(l, \alpha) \in \kappa_0 \cup \mathsf{new}(l, \alpha)$. Since ci is new, it is not present in w_0. By Definition 5, $w_0.\mathsf{new}\, ci : \kappa_0 \cup \mathsf{new}(\chi, \alpha), CE' \models ci : (\mathsf{chan}(\alpha), \chi)$.
By Lemma 6,$(w_0, \kappa_0, CE) \sqsubseteq (w_0.\mathsf{new}\, ci, \kappa_0 \cup \mathsf{new}(\chi, \alpha), CE')$.

$$\boxed{\textbf{Match Rule (5)} \ \frac{w_1' \ || \ w_2' \hookrightarrow w'}{w_1'.ci?val \ || \ w_2'.ci!val \hookrightarrow w'}}$$

If the evaluation respects match rule (5), the evaluation and typing at site s_1 must be of the following forms.

$$\frac{E \vdash_{s_1} e_1 \overset{w_1'}{\Longrightarrow} ci}{E \vdash_{s_1} e_1? \overset{w_1'.ci?val}{\Longrightarrow} val} \qquad \frac{\Gamma \vdash_{s_1} e_1 : (\mathsf{chan}(\bar{\tau}), \chi), \kappa}{\Gamma \vdash_{s_1} e_1? : (\bar{\tau}, \delta), \kappa \cup \mathsf{recv}(\chi, \bar{\tau})}$$

By induction hypothesis on e_1, there exists a CE' such that

$(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1', \kappa_0 \cup \kappa, CE')$ and $w_0.w_1' : \kappa_0 \cup \kappa, CE' \models ci : (\mathsf{chan}(\bar{\tau}), \chi)$.

By hypothesis we have that there is a matching evaluation at the remote site s_2 which must be of the following form

$$\frac{E \vdash_{s_2} e_1' \overset{w_1''}{\Longrightarrow} ci \quad E \vdash_{s_2} e_2' \overset{w_2''}{\Longrightarrow} val}{E \vdash_{s_2} e_1'!e_2' \overset{w_1''.w_2''.ci!val}{\Longrightarrow} val_{unit}}$$

where $w_2' = w_1''.w_2''$ and which must respect the following typing rule.

$$\frac{\Gamma \vdash_{s_2} e_1' : (\mathsf{chan}(\bar{\tau}'), \chi), \kappa_2 \quad \Gamma \vdash_{s_2} e_2' : (\bar{\tau}', \mu'), \kappa_2'}{\Gamma \vdash_{s_2} e_1'!e_2' : \mathsf{Unit}, \kappa_2 \cup \kappa_2' \cup \mathsf{send}(\chi, (\bar{\tau}', \mu))}$$

By induction hypothesis on e_1' and e_2', we can assume that

$(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1', \kappa_0 \cup \kappa, CE')$,
$w_0.w_2' : \kappa_0 \cup \kappa_2 \cup \kappa_2', CE' \models ci : \mathsf{chan}(\bar{\tau}'), \chi)$,
$w_0.w_2' : \kappa_0 \cup \kappa_2 \cup \kappa_2', CE' \models val : (\bar{\tau}', \mu')$.

By Definition of 5, it must be the case that $\bar{\tau} = \bar{\tau}'$. Then, Lemma 7,8 and 9 give respectively

$w_0.w_2'.ci!val : \kappa_0 \cup \kappa_2 \cup \kappa_2' \cup \mathsf{send}(\chi, \tau), CE' \models val : \tau$,
$w_0.w_1'.ci?val : \kappa_0 \cup \kappa \cup \mathsf{recv}(\chi, \bar{\tau}), CE' \models val : \tau$,
$(w_0, \kappa_0, CE) \sqsubseteq (w_0.w_1'.ci?val, CE')$. $\square$

C Soundness of Analysis

$\boxed{\textbf{case } \mathit{val} = \mathit{val}_{unit}}$

The proofs for the unit value and values of other constants are trivial and therefore are omitted.

$\boxed{\textbf{case } \mathit{val} = l@(n,s)}$

By hypothesis,

1. $w : \kappa, CE \models l@(n,s) : (\mathsf{chan}(\bar{\tau}), \chi)$ where $l \subseteq \chi$,
2. $\mathit{MChan} = \{l \mid (\mathsf{chan}, l) \in \mathit{Mobile}(l@(n,s))\} = \emptyset$
3. $\mathit{MFun} = \{l \mid (\mathsf{fn}, l) \in \mathit{Mobile}(l@(n,s))\} = \emptyset$

• If $\chi = \{l\}$ then by Rule (3) of Definition 12 we have $C = \{l\}, F = \emptyset$. Obviously, $\mathit{MChan} \subseteq C$ and $\mathit{MFun} \subseteq L$.

• If $\chi = \{l\} \cup \chi'$, the analysis must have followed Rule (6).

$$\frac{\vdash (\mathsf{chan}(\bar{\tau}), l) : C_1, F_1 \quad \vdash (\mathsf{chan}(\bar{\tau}), \chi') : C_2, F_2}{\vdash (\mathsf{chan}(\bar{\tau}), l \cup \chi') : C_1 \cup C_2, F_1 \cup F_2}$$

where $C_1 = l$ and $F_1 = \emptyset$. Obviously, $\mathit{MChan} \subseteq C_1$ and $\mathit{MFun} \subseteq F_1$. It follows that, $\mathit{MChan} \subseteq C_1 \cup C_2$ and $\mathit{MFun} \subseteq F_1 \cup F_2$.

$\boxed{\textbf{case } \mathit{val} = \langle l, E, x, e\rangle@s}$

By hypothesis,

1. $w : \kappa, CE \models \langle l, E, x, e\rangle@s : (\tau_1 \xrightarrow{\kappa'} \tau_2, \phi)$
   if there exists Γ such that $\Gamma \vdash_s \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa'} \tau_2, \phi), \emptyset$
   and $w : \kappa, CE \models E : \Gamma$
2. $\mathit{MChan} = \{l \mid (\mathsf{chan}, l) \in \mathit{Mobile}(\langle l, E, x, e\rangle@s)\}$
3. $\mathit{MFun} = \{l \mid (\mathsf{fn}, l) \in \mathit{Mobile}(\langle l, E, x, e\rangle@s)\}$

• If $\phi = \langle \varepsilon, l\rangle$ the analysis must have followed Rule (5).

$$\frac{\begin{array}{l} C_1 = \bigcup\{C \mid x \in \mathsf{Dom}(\varepsilon), \vdash \varepsilon(x) : C, F \text{ and } \varepsilon(x) \neq \langle \varepsilon, l\rangle\} \\ F_1 = \bigcup\{F \mid x \in \mathsf{Dom}(\varepsilon), \vdash \varepsilon(x) : C, F \text{ and } \varepsilon(x) \neq \langle \varepsilon, l\rangle\} \\ \vdash \tau_2 : C_2, F_2 \end{array}}{\vdash (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l\rangle) : C_1 \cup C_2, \{l\} \cup F_1 \cup F_2}$$

The hypothesis 1 above, due to Definitions 5 and 6, implies that

$w : \kappa, CE \models E(x) : \Gamma(x)$ for every $x \in Dom(E)$. Let $E' = Narrow(E)$. It follows that $w : \kappa, CE \models E'(x) : \varepsilon(x)$ for every $x \in \mathsf{Dom}(E')$ because ε is obtained by a restriction of Γ.

Let $L_{env} = \bigcup\{Mobile(E'(x')) \mid E' = Narrow(E, e, x), x' \in Dom(E'), x' \neq \langle l, E, x, e\rangle@s\}$. We can decompose L_{env} into two subsets such that $MChan'$ is the subset with elements tagged with chan and $MFun'$ is the subset with elements tagged with fn.

$$MChan' = \bigcup\{MChan_x \mid (\mathsf{chan}, l) \in L_{env}\}$$
$$MFun' = \bigcup\{MFun_x \mid (\mathsf{fn}, l) \in L_{env}\}$$

We can now apply the induction hypothesis on each element $MChan_x$ of $MChan'$ and C of C_1 to get $MChan' \subseteq C_1$. Similarly, for $MFun_x$ of $MFun'$ and F of F_1 to get $MFun' \subseteq F_1$. It follows that $MChan \subseteq C_1 \subseteq C_1 \cup C_2$ and $MFun \subseteq (\{l\} \cup F_1) \subseteq \{l\} \cup F_1 \cup F_2$.

• If $\phi = \langle \varepsilon, l\rangle \cup \phi'$ the analysis must have followed Rule (6).
The proof follows the similar reasoning as the above cases. □

Brief Biography

Zeliha Dilsun Kırlı received her B.Sc. in Computer Engineering from Middle East Technical University, Turkey and M.Sc. in Computer Science from the University of Edinburgh, UK. She is pursuing research towards a Ph.D. within the Laboratory for Foundations of Computer Science at the University of Edinburgh. Her interests include foundations of programming languages, mobile computation and program analysis.