# A Polymorphic Type and Effect System for Detecting Mobile Functions

Z. Dilsun Kırlı*

Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, UK.

**Abstract**

We define a language with a functional core and constructs for transmitting and receiving values across remote sites in which mobile agents have a natural representation as function closures. We present the dynamic semantics of this language in relational style. We develop a static type and effect system where the types are annotated so that the effects expose potentially mobile functions and channels. Finally, we prove the consistency of this system with respect to the dynamic semantics.

**Key words** 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

*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 [28, 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 [26] and the $\pi$-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 focus on the static estimation of potentially mobile functions and channels. We investigate this problem within the framework of a Facile-like language.

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 generate annotations automatically 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, transmission of a function occurs by copying of 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 [32]. 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.

Section 2 motivates the problem by an informal discussion and examples. Section 3 introduces the language on which our discussion is based. The type and effect system is defined in Section 4. Section 5 shows our approach to proving the soundness of our type and effect system. 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 annotating 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 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.

## 2.1 Mobile Functions

It should be relatively clear that any function which is passed directly to a 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 A Language for Communicating Expressions

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 a 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. We assume two disjoint sets of labels $L$ and $R$ which include labels for functions and channels respectively. The purpose of function labels ($l$) drawn from ($L$) and the channel labels ($r$) drawn from ($R$) is to uniquely identify functions and channels. Therefore, all labels in a program are required to be distinct.

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

## 3.2 Dynamic Semantics

### 3.2.1 Semantic Objects

Values of the language are the unit value, the values of base types such as integers and booleans, channel identifiers ($ci$) and function closures. The dynamic semantics distinguishes a set of values ($b$) as basic values.

Since the system consists of two expressions each of which return a value, we define a top-level value as a tupling of the individual values of expressions.

We also use semantic objects which describe the communication behaviour of an expression; $com$ stands for a communication event drawn from the set $\{ch\ ci, ci?val, ci!val\}$ representing the allocation of a channel, receiving and sending values over a channel respectively. The name $w$ stands for a sequence of communication events.

Other semantic objects include a channel identifier set $CI$ and an evaluation environment $E$ which is a mapping from identifiers to values.

$$
\begin{array}{lcl}
val & ::= & val_{unit} \mid val_{true} \mid val_{false} \mid val_n \\
 & & ci \mid \langle E, \texttt{fn}^l\, x \Rightarrow e \rangle \\
 & & \\
val_{top} & ::= & (val_1, val_2) \\
 & & \\
com & ::= & ch\, ci \mid ci?val \mid ci!val \\
 & & \\
w & ::= & \epsilon \mid com_1 \ldots com_n
\end{array}
$$

### 3.2.2 Top-level Rule and Matching

Our language does not include a construct for parallel composition of expressions. The concurrent activity is revealed in the composition of two sites. The top-level evaluation decomposes into individual evaluations at each site.

$$\frac{\begin{array}{l} CI, E_1 \vdash_{loc_1} e_1 \stackrel{w_1}{\Longrightarrow} val_1, CI_1 \quad CI, E_2 \vdash_{loc_2} e_2 \stackrel{w_2}{\Longrightarrow} val_2, CI_2 \\ w_1 \mid\mid w_2 \hookrightarrow w_{top} \quad w_{top} = ch\, ci_1 \ldots ch\, ci_n \quad (CI_1 \cup CI_2) \setminus CI = \{ci_1, \ldots ci_n\} \end{array}}{CI, (E_1, E_2), (loc_1[e_1] \mid\mid loc_2[e_2]) \stackrel{w_{top}}{\Longrightarrow} (val_1, val_2), CI_1 \cup CI_2}$$

The synchronisation is specified by the following matching rules between communication event sequences.

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

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

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

### 3.2.3 Sequential Evaluation Rules

A judgement of the form $CI, E \vdash e \stackrel{w}{\Longrightarrow} val, CI'$ states that expression $e$ evaluates to value $val$ against an environment $E$ and a global channel identifier set $CI$ updating it to $CI'$. The evaluation relation $\Longrightarrow$, which is defined in Figure 1, 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 only rule which updates the channel identifier set $CI$ is the rule for channel creation. This rule generates a new channel identifier which is globally unique. We can assume that each evaluation site generates fresh channel identifiers prefixed with its name. APPLY serves as a semantic tool to define the application of a predefined operation to an argument.

# 4 Type System

The type system specified in this section is designed with the intention of exposing which functions and channels an expression can possibly transmit. It extends the earlier type and effect systems which we mention in Section

| | |
|---|---|
| **(unit)** | $CI, E \vdash () \overset{\epsilon}{\Longrightarrow} val_{unit}, CI$ |
| **(true)** | $CI, E \vdash \texttt{true} \overset{\epsilon}{\Longrightarrow} val_{true}, CI$ |
| **(false)** | $CI, E \vdash \texttt{false} \overset{\epsilon}{\Longrightarrow} val_{false}, CI$ |
| **(int)** | $CI, E \vdash n \overset{\epsilon}{\Longrightarrow} val_n, CI$ |
| **(chan)** | $CI, E \vdash \texttt{chan}^r () \overset{ch\,ci}{\Longrightarrow} ci, CI \cup \{ci\}$ $\quad ci$ globally new |
| **(var)** | $CI, E \vdash x \overset{\epsilon}{\Longrightarrow} E(x), CI$ |
| **(fn)** | $CI, E \vdash \texttt{fn}^l\, x \Rightarrow e \overset{\epsilon}{\Longrightarrow} \langle E, \texttt{fn}^l\, x \Rightarrow e\rangle, CI$ |
| **(rec)** | $\dfrac{val = \langle E[f \mapsto val], \texttt{fn}^l\, x \Rightarrow e\rangle}{CI, E \vdash \texttt{rec}^l\, f(x) \Rightarrow e \overset{\epsilon}{\Longrightarrow} val, CI}$ |
| **(app)** | $\dfrac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} \langle E', \texttt{fn}^l\, x \Rightarrow e\rangle, CI' \quad CI', E \vdash e_2 \overset{w_2}{\Longrightarrow} val, CI'' \quad CI'', E'[x \mapsto val] \vdash e \overset{w_3}{\Longrightarrow} val', CI'''}{CI, E \vdash e_1\, e_2 \overset{w_1.w_2.w_3}{\Longrightarrow} val', CI'''}$ |
| **(primop)** | $\dfrac{CI, E \vdash \texttt{primop}\, c \overset{w_1}{\Longrightarrow} b, CI' \quad CI', E \vdash e \overset{w_2}{\Longrightarrow} val, CI'' \quad \text{APPLY}(b, val) = val_1}{CI, E \vdash \texttt{primop}\, c\, e \overset{w}{\Longrightarrow} val_1, CI''}$ |
| **(seq)** | $\dfrac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} val_1, CI' \quad CI', E \vdash e_2 \overset{w_2}{\Longrightarrow} val_2, CI''}{CI, E \vdash e_1\, ;\, e_2 \overset{w_1.w_2}{\Longrightarrow} val_2, CI''}$ |
| **(if-t)** | $\dfrac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} val_{true}, CI' \quad CI', E \vdash e_2 \overset{w_2}{\Longrightarrow} val, CI''}{CI, E \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \overset{w_1.w_2}{\Longrightarrow} val, CI''}$ |
| **(if-f)** | $\dfrac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} val_{false}, CI' \quad CI', E \vdash e_3 \overset{w_2}{\Longrightarrow} val, CI''}{CI, E \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \overset{w_1.w_2}{\Longrightarrow} val, CI''}$ |
| **(let)** | $\dfrac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} val_1, CI' \quad CI', E[x \mapsto val_1] \vdash e_2 \overset{w_2}{\Longrightarrow} val_2, CI''}{CI, E \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \overset{w_1.w_2}{\Longrightarrow} val_2, CI''}$ |
| **(receive)** | $\dfrac{CI, E \vdash e \overset{w}{\Longrightarrow} ci, CI'}{CI, E \vdash e? \overset{w.ci?val}{\Longrightarrow} val, CI'}$ |
| **(send)** | $\dfrac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} ci, CI' \quad CI', E \vdash e_2 \overset{w_2}{\Longrightarrow} val, CI''}{CI, E \vdash e_1 ! e_2 \overset{w_1.w_2.ci!val}{\Longrightarrow} val_{unit}, CI''}$ |

Figure 1: Evaluation Rules

6 so that the types capture additional information in the form of abstract closure annotations.

## 4.1 Semantic Objects

We have abstract closures ($v$), channel regions ($\chi$), communication effects ($\kappa$), raw types ($\bar{\tau}$) and closure annotated types ($\tau$). A type scheme ($\sigma$) can be obtained from a type ($\tau$) by universally quantifying zero or more abstract closure, region, behaviour and type variables.

$$
\begin{array}{lcl}
v & ::= & \emptyset \mid \langle \varepsilon, l \rangle \mid \gamma \mid v \cup v \\
\chi & ::= & r \mid \rho \mid \chi \cup \chi \\
\kappa & ::= & \emptyset \mid ch(\chi, \tau) \mid send(\chi, L) \mid remref(\chi, R) \mid recv(\chi, \tau) \mid \beta \mid \kappa \cup \kappa \\
\bar{\tau} & ::= & Unit \mid Int \mid Bool \mid chan_{\chi}(\tau) \mid \tau \xrightarrow{\kappa} \tau \mid \alpha \\
\tau & ::= & (\bar{\tau}, v) \\
\sigma & ::= & \forall \vec{\gamma} \vec{\rho} \vec{\beta} \vec{\alpha}.\, \tau
\end{array}
$$

The set $v$ conservatively estimates the set of functions an expression can evaluate to. A function closure $\langle E, \mathtt{fn}^{l}\, x \Rightarrow e \rangle$ is abstracted by $\langle \varepsilon, l \rangle$ where $\varepsilon$ stands for the environment of the function and $l$ stands for the function. The abstract environment $\varepsilon$ contains static bindings of free variables of the function expression with function or channel types. The meta-variable $\gamma$ ranges over abstract closures. The operator $\cup$ denotes set union.

The set $\chi$ estimates channel regions. A more detailed explanation of regions can be found in [4, 18]. Channel regions are to channels what abstract closures are to functions. They abstract channels and thus provide a means of keeping track of the identity of channels in the static semantics. A channel region is represented by a label $r$. The meta-variable $\rho$ ranges over channel regions.

The set $\kappa$ estimates the communication effects which may occur during the evaluation of an expression. An expression may have no effect, or its overall effect can be estimated by a set of possible effects; allocating a channel, sending a function closure approximated by the set $L$ or creating a remote reference to a channel which is approximated by the set $R$ through a channel and receiving a value on a channel. The meta-variable $\beta$ ranges over communication effects.

Raw types ($\bar{\tau}$) consist of base types, channel types, function types and type variables ($\alpha$). We pair raw types with abstract closure annotations to get the types of our language. An expression of type $chan_{\chi}(\tau)$ denotes a channel which communicates values of type $\tau$ where $\chi$ estimates the channel. An expression of type $(\bar{\tau}_1, v_1) \xrightarrow{\kappa} (\bar{\tau}_2, v_2)$ denotes a function whose argument

is estimated by $v_1$ and whose result is estimated by $v_2$. By injecting value annotations into types we keep track of functions flowing as arguments and results. The annotation $\kappa$ stands for the latent effect which may occur when the function is applied.

### 4.1.1 Operations on Types

The operations defined below are familiar from polymorphic type systems for languages with a functional core.

**Definition 1 ($\succ$)**

A type scheme $\sigma$ generalises a type $\tau$, written as $\sigma \succ \tau$, if $\tau$ can be obtained from the body of $\sigma$ by substituting types ($\tau$), behaviours ($\kappa$), regions ($\chi$) and abstract closures ($v$) for the bound type, behaviour, region and abstract closure variables of $\sigma$ respectively. A substitution $\theta$ denotes a mapping from type, behaviour, region and abstract closure variables to types, behaviours, regions and abstract closures.

**Definition 2 (Generalisation)**

Types are generalised to type schemes by the operation *Gen*.

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

where $fv$ computes the free type, behaviour, region and abstract closure variables in the expected way.

### 4.1.2 Analysis for Mobility

The following two definitions are related to forming and analysing abstract closures and therefore are characteristic operations of our type system.

**Definition 3 (Abstract Closure)**

$AbsCl(\Gamma, e, S) = \varepsilon$ where $\varepsilon$ is the environment $\Gamma$ restricted to the bindings of free identifiers of $e$ excluding the ones in set $S$, which have function or channel types.

The information with respect to function closures embodied in types through $v$ annotations are analysed to extract the labels of potentially mobile functions and channels. The rules of this analysis are specified by an inference system. A judgement of the form $\vdash_{mob} \tau : R, L$ means that the transmission of a value of type $\tau$ may cause the transmission of channels with labels in $R$ and functions with labels in $L$.

**Definition 4** ( $\vdash_{mob}$ )

**(1)** $\vdash_{mob} (\bar{\tau}, \emptyset) : \emptyset, \emptyset \quad \bar{\tau}$ base type or variable

**(2)** $\vdash_{mob} (chan_{\rho}(\tau), \emptyset) : \emptyset, \emptyset$

**(3)** $\vdash_{mob} (chan_{r}(\tau), \emptyset) : \{r\}, \emptyset$

**(4)** $$\frac{\vdash_{mob} (chan_{\chi_1}(\tau), \emptyset) : R_1, L_1 \quad \vdash_{mob} (chan_{\chi_2}(\tau), \emptyset) : R_2, L_2}{\vdash_{mob} (chan_{\chi_1 \cup \chi_2}(\tau), \emptyset) : R_1 \cup R_2, L_1 \cup L_2}$$

**(5)** $$\frac{\vdash_{mob} \tau_2 : R_2, L_2}{\vdash_{mob} (\tau_1 \xrightarrow{\kappa} \tau_2, \gamma) : R_2, L_2}$$

**(6)** $$\frac{\begin{array}{l}\text{if } \varepsilon(x) = (\tau_x) \quad \vdash_{mob} \tau_x : R_x, L_x \\ R_1 = \bigcup R_x, L_1 = \bigcup L_x \\ \vdash_{mob} \tau_2 : R_2, L_2\end{array}}{\vdash_{mob} (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle) : R_1 \cup R_2, \{l\} \cup L_1 \cup L_2}$$

**(7)** $$\frac{\vdash_{mob} (\bar{\tau}, v_1) : R_1, L_1 \quad \vdash_{mob} (\bar{\tau}, v_2) : R_2, L_2}{\vdash_{mob} (\bar{\tau}, v_1 \cup v_2) : R_1 \cup R_2, L_1 \cup L_2}$$

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.

The Rules 2,3,4 apply when the type is a channel type and the channel is estimated by a region variable, single region or a set of regions respectively. A variable includes no specific information, hence the empty $R$ in Rule 2. In cases where labels are present in the set $\chi$, Rules 3 and 4 collect these one by one and merge them in $R$.

The Rules 5,6,7 apply when the type is a function type. If an abstract closure 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 5 handles this case. If the function is abstracted by a single abstract closure, Rule 6 extracts the label of the function and examines the bindings of the identifiers of the abstract environment to collect the labels of potentially mobile values. The return type is also examined to handle the escaping values. If the function is estimated by a non-singleton set Rule 7 ensures that each element of the set is examined and the results are merged into a single set.

| | |
|---|---|
| **(unit)** | $\Gamma \vdash () : (Unit, \emptyset), \emptyset$ |
| **(true)** | $\Gamma \vdash \texttt{true} : (Bool, \emptyset), \emptyset$ |
| **(false)** | $\Gamma \vdash \texttt{false} : (Bool, \emptyset), \emptyset$ |
| **(int)** | $\Gamma \vdash n : (Int, \emptyset), \emptyset$ |
| **(chan)** | $\Gamma \vdash \texttt{chan}^r() : (chan_\chi(\tau), \emptyset), ch(\chi, \tau) \quad \tau = (\alpha, \gamma)\,, \alpha, \gamma \text{new}\,,\ r \subseteq \chi$ |
| **(var)** | $\dfrac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau}$ |
| **(fn)** | $\dfrac{\Gamma[x \ \mapsto \ \tau_1] \vdash e : \tau_2, \kappa \quad \varepsilon = AbsCl(\Gamma, e, \{x\})}{\Gamma \vdash \texttt{fn}^l\, x \ \Rightarrow e : (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle), \emptyset}$ |
| **(rec)** | $\dfrac{\Gamma[f \ \mapsto \ (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle)][x \ \mapsto \ \tau_1] \vdash e : \tau_2, \kappa \quad \varepsilon = AbsCl(\Gamma, e, \{x, f\})}{\Gamma \vdash \texttt{rec}^l\, f\,(x) \Rightarrow e : (\tau_1 \xrightarrow{\kappa} \tau_2, \langle \varepsilon, l \rangle), \emptyset}$ |
| **(app)** | $\dfrac{\Gamma \vdash e_1 : ((\bar{\tau}_1, v) \xrightarrow{\kappa} (\bar{\tau}_2, v_2), v'), \kappa' \quad \Gamma \vdash e_2 : (\bar{\tau}_1, v_1), \kappa'' \quad v_1 \subseteq v}{\Gamma \vdash e_1\, e_2 : (\bar{\tau}_2, v_2), \kappa \cup \kappa' \cup \kappa''}$ |
| **(primop)** | $\dfrac{\Gamma \vdash \texttt{primop}\, c : ((\bar{\tau}_1, v) \xrightarrow{\kappa} (\bar{\tau}_2, v_2), v'), \emptyset \quad \Gamma \vdash e : (\bar{\tau}_1, v_1), \kappa' \quad v_1 \subseteq v}{\Gamma \vdash \texttt{primop}\, c\, e : (\bar{\tau}_2, v_2), \kappa \cup \kappa'}$ |
| **(seq)** | $\dfrac{\Gamma \vdash e_1 : \tau_1, \kappa \quad \Gamma \vdash e_2 : \tau_2, \kappa'}{\Gamma \vdash e_1\ ;\ e_2 : \tau_2, \kappa \cup \kappa'}$ |
| **(if)** | $\dfrac{\Gamma \vdash e_1 : (Bool, \emptyset), \kappa, \quad \Gamma \vdash e_2 : (\bar{\tau}, v), \kappa' \quad \Gamma \vdash e_3 : (\bar{\tau}, v'), \kappa''}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : (\bar{\tau}, v \cup v'), \kappa \cup \kappa' \cup \kappa''}$ |
| **(let)** | $\dfrac{\Gamma \vdash e_1 : \tau_1, \kappa \quad \Gamma[x \ \mapsto \ Gen(\kappa, \Gamma)(\tau_1)] \vdash e_2 : \tau_2, \kappa'}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2, \kappa \cup \kappa'}$ |
| **(receive)** | $\dfrac{\Gamma \vdash e : (chan_\chi(\tau), \emptyset), \kappa}{\Gamma \vdash e? : \tau, \kappa \cup recv(\chi, \tau)}$ |
| **(send)** | $\dfrac{\Gamma \vdash e_1 : (chan_\chi(\tau), \emptyset), \kappa \quad \Gamma \vdash e_2 : \tau, \kappa' \quad \vdash_{mob} \tau : R, L}{\Gamma \vdash e_1 ! e_2 : Unit, \kappa \cup \kappa' \cup send(\chi, L) \cup remref(\chi, R)}$ |
| **(subs)** | $\dfrac{\Gamma \vdash e : (\bar{\tau}, v), \kappa \quad \kappa \subseteq \kappa' \quad v \subseteq v'}{\Gamma \vdash e : (\bar{\tau}, v'), \kappa'}$ |

Figure 2: Typing Rules

### 4.1.3 Typing Rules

The static semantics for the language assigns a type, an effect $(\tau, \kappa)$ to each expression. This is represented by a judgement of the form $\Gamma \vdash e : \tau, \kappa$ .

The context in which an expression is assigned a type, an effect and an abstract value is represented by a static environment $\Gamma$, which maps value identifiers to type schemes $(\sigma)$. The notation $\Gamma[x \mapsto \sigma]$ is used for adding element $x$ to the environment $\Gamma$, overriding the existing binding if $x$ is already in the domain of $\Gamma$.

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 **(send)** for the send expression reveals the essence of the type system. All the information with respect to the sent value captured by its type is analysed to extract the labels of the channels and the closures which are potentially mobile.

The side condition in the application rule **(app)** is used to ensure that the abstract closure annotation inferred for the argument is consistent with the abstract closure annotation which appears in the argument position of the function type. The operator $\subseteq$ denotes the subset relation for sets.

The subsumption rule **(subs)** allows an expression to admit a larger communication effect and an abstract closure than seems to be necessary. We adopt the approach which is referred to as early subsumption in [18] and subeffecting in [32]. 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.

# 5 Semantic Consistency

We define the following consistency judgements which relate 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 mapping from channel identifiers to channel region and type pairs: $[ci_1 \mapsto (\chi_1, \tau), \ldots, ci_n \mapsto (\chi_n, \tau)]$. The rest of the objects we refer to are as defined in the previous sections.

| | |
|---|---|
| $CE \models val : \tau$ | the value $val$ has the type $\tau$ |
| $CE \models E : \Gamma$ | the values of $E$ respect the types of $\Gamma$ |
| $CE \models com : \kappa$ | the event $com$ respects the effect $\kappa$ |
| $CE \models w : \kappa$ | the event sequence $w$ respects the effect $\kappa$ |

The definition of $\models$ makes use of an ordering relation $\preceq$ between abstract closures which is defined as follows.

**Definition 5 ($\preceq$)**

| | |
|---|---|
| $\epsilon \preceq v$ | for any $v$ |
| $\langle \varepsilon_1, l \rangle \preceq v$ | if there exists $\langle \varepsilon_2, l \rangle \in v$ such that $\varepsilon_1 \subseteq \varepsilon_2$ |

The consistency relation which establishes the correspondence between objects of the dynamic semantics and the static semantics satisfies the following properties:

**Definition 6 ($\models$)**

$CE \models val_{unit} : (Unit, \epsilon)$
$CE \models val_{true} : (Bool, \epsilon)$
$CE \models val_{false} : (Bool, \epsilon)$
$CE \models val_n : (Int, \epsilon)$

$CE \models ci : (chan_\chi(\tau), \epsilon)$
if $ch(\chi, \tau) \in \kappa, CE(ci) = (\chi, \tau)$
$CE \models \langle E, \mathtt{fn}^l\, x \Rightarrow e \rangle : (\tau_1 \xrightarrow{\kappa'} \tau_2, v)$
if there exists $\Gamma$ such that $\Gamma \vdash \mathtt{fn}^l\, x \Rightarrow e : (\tau_1 \xrightarrow{\kappa'} \tau_2, v'), \emptyset$
where $v \preceq v'$ and $CE \models E : \Gamma$

$CE \models E : \Gamma$
if $Dom(E) = Dom(\Gamma)$ and for any $x \in Dom(E)$ $CE \models E(x) : v$
such that $v \preceq \Gamma(x)$.
$CE \models \epsilon : \emptyset$
$CE \models ch\, ci : ch(\chi, \tau)$
if $CE(ci) = (\chi, \tau)$
$CE \models ci?val : recv(\chi, \tau)$
if $CE(ci) = (\chi, \tau)$
$CE \models ci!val : send(\chi, L) \cup remref(\chi, R)$
if $CE(ci) = (\chi, \tau)$ and $CE \models val : \tau$
and $\vdash_{mob} \tau : R', L'$ such that $R' \subseteq R, L' \subseteq L$
$CE \models w : \kappa$
if there exists a $\kappa_i \in \kappa$ such that $CE \models com : \kappa_i$ for any $com \in w$

We write $\Gamma \models val : \forall \vec{\iota}.\tau$ if and only if $\Gamma \models val : \theta\tau$ for any substitution $\theta$ defined on $\vec{\iota}$.

The typing consistency relation that we seek to define must be viewed as the maximal fixed point of the property defined above. We consider an appropriate function $F$ and by verifying that $F$ is monotonic conclude that the above definition is satisfied by its maximal fixed point. This part of the proof is omitted from our discussion.

**Definition 7 (Extension)**

$(\kappa', CE')$ extends $(\kappa, CE)$, 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 8 (Equivalence)**

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

**Definition 9 (Succession)**

$(CI, w, \kappa, CE) \sqsubseteq (CI', w', \kappa', CE')$ if and only if $CI \subseteq CI', w' = w.w''$ for some $w''$, $(\kappa, CE) \sqsubseteq (\kappa', CE')$,$CE \models val : \tau$ and $CE \models w : \kappa$ implies $CE' \models val : \tau$ and $CE' \models w' : \kappa'$.

**Proposition 1 (Consistency)**

Assume $CI, (E_1, E_2)(loc_1[e_1] \ || \ loc_2[e_2]) \overset{w_{top}}{\Longrightarrow} (val_1, val_2), CI_{top}$.
For each $e \in \{e_1, e_2\}$, if $CE \models E : \Gamma$ and $CE \models w_0 : \kappa_0$, $\Gamma \vdash e : (\bar{\tau}, v), \kappa$ and $CI, E \vdash e \overset{w}{\Longrightarrow} val, CI'$ then there exists a $CE'$ and $v'$ such that $CE' \models val : \bar{\tau}, v'$ and $(CI, w_0, \kappa_0, CE \sqsubseteq CI', w_0.w, \kappa_0 \cup \kappa, CE')$ and $v' \preceq v$.

**Proof (Consistency)** The proof is given by a simultaneous induction on the depth of inference of $CI, E_i \vdash_{loc_i} e_i \overset{w_i}{\Longrightarrow} val_i, CI'$ for $i \in \{0, 1\}$. □

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, 30, 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 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 [29]. 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 [27] 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 the functions which play the key role. Therefore, we introduce the concept of abstract closure which can be considered as an analogous concept to channel regions. We retain channel regions in our type system because of two reasons. Firstly, we would like our type system to be at least as expressive as the earlier ones. Secondly, we are interested in the potential mobility of the channels as well as that of functions.

# 7 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 $\pi$-calculus family and static analyses techniques developed for these [9, 24, 10, 31, 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 CML 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 $\pi$-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] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College, London University, 1989.

[27] 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.

[28] 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.

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

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

[31] 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.

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

# A Lemmas

**Lemma 1 (General Substitution)**

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

**Proof (General Substitution)** The proof is given by induction on the derivation of $\Gamma \vdash e : \tau, \kappa$.

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

Suppose $\sigma = \forall\vec{\iota}.\tau_0$. After renaming if necessary we define $\theta$ so that $\vec{\iota}$ is out of reach of $\theta$. Let $\theta'$ be a substitution over $\vec{\iota}$ such that $\theta'(\tau_0) = \tau$. Now define a substitution $\theta''$ 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 $\beta$ 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, $\theta(\Gamma) \vdash x : \Gamma(\tau)$.

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

Let $\forall\vec{\iota}.\tau_1$ be $Gen(\kappa,\Gamma)(\tau_1)$. For any substitution $\theta$, let us consider fresh $\vec{\iota'}$ and define $\theta'$ 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 $\theta'$, $\iota$ 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 $\theta'$ we get $\theta'\Gamma \vdash e : \theta'\tau, \theta'\kappa$. By definition of $\theta'$, $\theta'\Gamma \vdash e : \theta'\tau, \theta\kappa$. By using the induction hypothesis on $e_2$ we get $\theta(\Gamma[x \mapsto Gen(\kappa,\Gamma)(\tau_1)]) \vdash e_2 : \theta(\tau_2), \theta(\kappa')$ which is equivalent to $\theta(\Gamma)[x \mapsto Gen(\theta(\kappa),\theta(\Gamma))(\theta'(\tau_1))] \vdash e_2 : \theta(\tau_2), \theta(\kappa')$. Finally by definition of **(let)** we can conclude that $\theta(\Gamma) = \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \theta(\tau), \theta(\kappa)$. □

**Lemma 2 (Extension)**

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

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

**Lemma 3 (Equivalence)**

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

**Proof (Equivalence)** Since $(\kappa, CE) \simeq (\kappa', CE')$, by Definition 7,
$(\kappa, CE) \sqsubseteq (\kappa', CE')$ and $(\kappa', CE') \sqsubseteq (\kappa, CE)$. If $CE \models val : \tau$ and $CE \models w : \kappa$ then by Lemma 2, $CE' \models val : \tau$ and $CE' \models w : \kappa$. If $S : CE' \models val : \tau$ and $CE' \models w : \kappa$ then by Lemma 2, $CE \models val : \tau$ and $CE \models w : \kappa$. □

**Lemma 4 (Semantic Substitution)**

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

**Proof (Semantic Substitution)** We consider $Z = \{(w, \theta\kappa, \theta CE, val, \theta\tau) \mid CE \models val : \tau$ and $CE \models w : \kappa\}$ and show by case analysis on the structure of $val$ that $q = (w, \theta\kappa, \theta CE, val, \theta\tau)$ is in $F(Z)$. □

**Lemma 5 (Instantiation)**

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

**Proof (Instantiation)** By Lemma 4, $\theta CE \models val : \theta\tau$ and $CE \models w : \theta\kappa$. Since $\theta\kappa = \kappa$, by Definition 7, $(\kappa, CE) \simeq (\kappa, \theta CE)$. By Lemma 3 we conclude that $CE \models val : \theta\tau$ and $CE \models w : \kappa$. □

**Lemma 6 (Channel allocation)**

Let $\kappa' = ch(\chi, \tau)$ , $w' = w.ch\, ci$ and $CE' = CE[ci \mapsto (\chi, \tau)]$ where $ci \notin Dom(CE)$. If $CE \models val : \tau'$ and $CE \models w : \kappa$ then $CE' \models val : \tau'$ and $CE' \models w' : \kappa \cup \kappa'$.

**Proof (Channel Allocation)** We consider the set $Z = \{(w', \kappa \cup \kappa', CE', val, \tau') \mid CE \models val : \tau, CE \models w : \kappa\}$. We show by case analysis on the structure of $val$ that $q = (w', \kappa \cup \kappa', CE', val, \tau')$ is in $F(Z)$. □

**Lemma 7 (Send)**

Let $\kappa' = send(\chi, L) \cup remref(\chi, R), w' = w.ci!val, CE \models ci : (chan_\chi(\tau), \epsilon)$, $CE \models w : \kappa, CE \models val : \tau$ and $\vdash_{mob} \tau : R', L'$ where $R' \subseteq R, L' \subseteq L$ . If $CE \models val' : \tau'$ and $CE \models w : \kappa$ then $CE' \models val' : \tau'$ and $CE \models w' : \kappa \cup \kappa'$.

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

# B Proof of consistency

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

$$CI, E \vdash \texttt{chan}^r() \overset{ch\, ci}{\Longrightarrow} ci, CI \cup \{ci\} \;\; ci \notin CI$$

$\Gamma \vdash \texttt{chan}^r() : (chan_\chi(\tau), \emptyset), ch(\chi, \tau)$ where $\tau = (\alpha, \gamma), \alpha, \gamma$ are new, $r \subseteq \chi$ . By hypothesis $CE \models w_0 : \kappa_0$. We are required to show that there exits $CE'$ and $v'$ such that $CE' \models val : \bar{\tau}, v'$ and $(CI, w_0, \kappa_0, CE \sqsubseteq CI', w.ch\, ci, \kappa_0 \cup ch(\chi, \tau), CE')$ and $v' \preceq v$.

We provide a witness for $CE'$ by taking $CE' = CE[ci \mapsto (\chi, \tau)]$ such that $ci \notin CI$ and $ci \notin Dom(CE)$. By Lemma 6 we know that if $CE \models val : \tau'$ then $CE' \models val : \tau'$ and $CE' \models w.chan\, ci : \kappa_0 \cup ch(\chi, \tau)$. Now, by Definition

5 we also have $(CI, w, \kappa, CE \sqsubseteq CI', w.chan\, ci, \kappa \cup ch(\chi, \tau), CE')$. We provide a witness for $v'$ by taking $v' = \epsilon$.

$\boxed{\textbf{case } e_1 e_2}$ The rules **(app)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$\frac{CI, E \vdash e_1 \overset{w_1}{\Longrightarrow} \langle E', \mathtt{fn}^l\, x \Rightarrow e\rangle, CI' \quad CI', E \vdash e_2 \overset{w_2}{\Longrightarrow} val, CI'' \quad CI'', E'[x \mapsto val] \vdash e \overset{w_3}{\Longrightarrow} val', CI'''}{CI, E \vdash e_1\, e_2 \overset{w_1.w_2.w_3}{\Longrightarrow} val', CI'''}$$

$$\frac{\Gamma \vdash e_1 : ((\bar{\tau}_1, v) \overset{\kappa}{\rightarrow} (\bar{\tau}_2, v_2), v'), \kappa' \quad \Gamma \vdash e_2 : (\bar{\tau}_1, v_1), \kappa'' \quad v_1 \subseteq v}{\Gamma \vdash e_1\, e_2 : (\bar{\tau}_2, v_2), \kappa \cup \kappa' \cup \kappa''}$$

By the induction hypothesis on $e_1$ and the assumption $CE \models E' : \Gamma$ and $CE \models w_0 : \kappa_0$, there exists $CE'$ and $v''$ such that
$CE' \models \langle E', \mathtt{fn}^l\, x \Rightarrow e\rangle : ((\bar{\tau}_1, v) \overset{\kappa}{\rightarrow} (\bar{\tau}_2, v_2), v'')$, $v'' \preceq v'$ and
$(CI, w_0, \kappa_0, CE) \sqsubseteq (CI', w.w_1, \kappa_0 \cup \kappa', CE')$. By Definition 5 we can conclude that $CE' \models E' : \Gamma$ and $CE' \models w_0.w_1 : \kappa_0 \cup \kappa'$. Now, by induction hypothesis on $e_2$ there exists $CE''$ and $v'''$ such that $CE'' \models val : (\bar{\tau}_1, v''')$ , $v''' \preceq v_1$ and $(CI', w_0, \kappa_0 \cup \kappa', CE') \sqsubseteq (CI'', \kappa_0 \cup \kappa' \cup \kappa'', CE'')$. By Definition 5, since $CE' \models \langle E', \mathtt{fn}^l\, x \Rightarrow e\rangle : ((\bar{\tau}_1, v) \overset{\kappa}{\rightarrow} (\bar{\tau}_2, v_2), v'')$, we have $CE'' \models \langle E', \mathtt{fn}^l\, x \Rightarrow e\rangle : ((\bar{\tau}_1, v) \overset{\kappa}{\rightarrow} (\bar{\tau}_2, v_2), v'')$ and $CE'' \models w_0.w_1 : \kappa_0 \cup \kappa' \cup \kappa''$. $CE'' \models \langle E', \mathtt{fn}^l\, x \Rightarrow e\rangle : ((\bar{\tau}_1, v) \overset{\kappa}{\rightarrow} (\bar{\tau}_2, v_2), v'')$ requires that there exists a $\Gamma'$ such that $\Gamma' \vdash \mathtt{fn}^l\, x \Rightarrow e : ((\bar{\tau}_1, v) \overset{\kappa}{\rightarrow} (\bar{\tau}_2, v_2), v''''), \emptyset$ where $v'' \preceq v''''$ and $CE'' \models E' : \Gamma'$.

Let us take $E'' = E'[x \mapsto val]$ and $\Gamma'' = \Gamma'[x \mapsto (\bar{\tau}_1, v)]$. By the side condition $v_1 \subseteq v$ and Definition $(\preceq)$ , it follows that $v''' \preceq v$. By Definition $(\models)$ and Lemma 2 we have $CE'' \models E'' : \Gamma''$ and $CE'' \models w_0.w_1.w_2 : \kappa_0 \cup \kappa' \cup \kappa''$. Now, by induction hypothesis on $e_3$, we can conclude that there exists a $CE'''$ and $v_2'$ such that $v_2' \preceq v_2$ and $(CI'', w_0.w_1.w_2, \kappa_0 \cup \kappa' \cup \kappa'', CE'') \sqsubseteq (CI''', , w_0.w_1.w_2.w_3, \kappa_0 \cup \kappa' \cup \kappa'' \cup \kappa, CE''')$.

$\boxed{\textbf{case } e_1!e_2}$ The rules **(send)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$\frac{CI, E \vdash e_1 \stackrel{w_1}{\Longrightarrow} ci, CI' \quad CI', E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val, CI''}{CI, E \vdash e_1!e_2 \stackrel{w_1.w_2.ci!val}{\Longrightarrow} val_{unit}, CI''}$$

$$\frac{\Gamma \vdash e_1 : (chan_\chi(\tau), \emptyset), \kappa \quad \Gamma \vdash e_2 : \tau, \kappa' \quad \vdash_{mob} \tau : R, L}{\Gamma \vdash e_1!e_2 : Unit, \kappa \cup \kappa' \cup send(\chi, L) \cup remref(\chi, R)}$$

By the induction hypothesis on $e_1$ and the assumption $CE \models E : \Gamma$ and $CE \models w_0 : \kappa_0$ there exists $CE'$ and $v'$ such that $CE' \models ci : chan_\chi(\bar{\tau}, \epsilon)$, $(CI, w_0, \kappa_0, CE) \sqsubseteq (CI', w_0.w_1, \kappa_0 \cup \kappa, CE')$ and $\epsilon \preceq \emptyset$.

By Definition 5 we have $CE' \models E : \Gamma$ and $CE' \models w_0.w_1 : \kappa_0 \cup \kappa$. We can now apply the induction hypothesis on $e_2$. It follows that there exists $CE''$ and $v''$ such that $CE'' \models val : (\bar{\tau}, v'')$ where $\tau = (\bar{\tau}, v)$ and $(CI', w_0.w_1, \kappa_0 \cup \kappa', CE') \sqsubseteq (CI'', w_0.w_1.w_2, \kappa_0 \cup \kappa \cup \kappa', CE'')$ and $v'' \preceq v$. By Definition 5 we also have $CE' \models ci : chan_\chi(\tau), v'$.

The side condition imposes that $\vdash_{mob} \tau : R, L$. We know that $\tau = (\bar{\tau}, v)$ and $v'' \preceq v$. Suppose $\vdash_{mob} (\bar{\tau}, v'') = R', L'$. By inspecting the definition of $\vdash_{mob}$ it follows that $R' \subseteq R, L' \subseteq L$. By Lemma 7 we can conclude that $(CI'', w_0.w_1.w_2, \kappa_0 \cup \kappa \cup \kappa', CE'' \sqsubseteq (CI'', w_0.w_1.w_2.ci!val, \kappa_0 \cup \kappa \cup \kappa' \cup send(\chi, L) \cup remref(\chi, R), CE''), CE'')$.

Finally, by Definition 5 and $\models$ we conclude that $CE'' \models val_{unit} : (Unit, \epsilon)$ and $(CI, w_0, \kappa_0, CE) \sqsubseteq (CE'', \kappa_0 \cup \kappa' \cup \kappa'' \cup send(\chi, L) \cup remref(\chi, R), CE'')$. $\square$