A Static Type System for Detecting Potentially Transmissible Functions

Dilsun Kırlı

LFCS, Edinburgh

April 9, 1999

Abstract

We focus on 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 propose a type system enhanced with effects and a control flow analysis component to statically detect the potential transmissibility of functions.

1 Introduction

It has been claimed by several researchers that prominent features of sequential functional languages such as having functions as first class values, robust type systems and automatic memory management are also desirable features for concurrent programming languages. This claim has been supported by implementations of languages such as Concurrent ML, Poly/ML, Facile [1] and Concurrent Haskell [2] which are functional languages enhanced with facilities for transmitting and receiving values over typed channels.

The emergence of the mobile computation paradigm has now opened up a new avenue for investigating how well the benefits of features from functional programming languages carry over to programming languages for mobile computation. Knabe's Facile [3] constitutes a good example for this. It takes the original Facile system [4] as its departure point and by reusing several parts of its design and implementation arrives at a language which is particularly well-suited for the construction of mobile agents.

The Facile language combines Standard ML [5] with a model of concurrency based on CCS and its higher-order mobile extensions (CHOCS and the π-calculus) [6, 7] and has a well-defined semantics [8]. This makes Facile an apt choice to use as a basis for obtaining insights into the problem of integrating functional and mobile computation.

It is demonstrated by the work of [3] that the first-class nature of functions would prove to be very convenient for representing mobile agents if one could successfully deal with a key issue in mobile computation, namely the heterogeneity of network nodes. By heterogeneity we mean that different nodes of a network may be of different architectures and therefore support different value representations. These differences must be taken into consideration by language implementations which facilitate transmission of values across the network.

A common method for dealing with heterogeneity is to adopt a standard transmissible representation. Before transmission, each value is converted into a standard representation determined by its type. This operation is called *marshalling*. Upon receipt, the standard representation is converted into the representation appropriate for the receiving machine in an operation called *unmarshalling*. Marshalling is a recursive process which decomposes a value until it terminates with the conversion of primitive components such as integers and characters.

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. The primitive component of a closure in Facile is the machine code of the function. Converting machine code to a standard representation is a difficult problem and the remaining possibility of translating one native machine code into another one at run-time is mostly impractical.

The fact that functions produced at run-time are just new closures containing old code makes it possible to circumvent this impracticality. One can perform marshalling and generate transmissible representations of code at compile-time and an increased run-time performance can be obtained. Note that other data values which we can only know at run-time still require run-time marshalling.

Generating the transmissible representation for each function and storing them, however would be very space inefficient. Different approaches to reduce the space cost have been discussed by [3]. The approach adopted by Facile is to perform compile-time marshalling only for those functions which have been explicitly annotated by the programmer to be potentially transmissible.

The identification of potentially transmissible functions automatically without resorting to user annotations has been stated as future work in the dissertation of [3]. In this paper, we tackle this problem by proposing a type system enhanced with effects and a control flow analysis component. Section 2 motivates the problem by examples and gives an overview of the related work. Sections 3 introduces the language on which our discussion is based. The proposed type system is defined in section 4 and is proved to be consistent with respect to the dynamic semantics of the language. The following sections discuss the direction of our on-going work.

2 Potentially Transmissible Functions

Knabe determines two criteria to guide programmers in annotating potentially transmissible functions in their programs.

- Any function which is passed directly to a send operator needs to be annotated.
- Any function which is referred to by an annotated function and is not defined within it needs to be annotated. (Any function is a part of the code of the function within which it is defined. Therefore it need not be annotated if the embodying function is already annotated).

If the above criteria were sufficient, all the information needed for automatic detection of transmissible 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, these two criteria are not sufficient in the presence of higher-order functions. One needs to go beyond the information provided by the environment component of the function closure.

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 Examples

The following examples are inspired by a discussion in Knabe's dissertation.

Example 1

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

Obviously, g needs to be annotated. However, the above criteria can no longer be taken literally for h. The function h is referred to by g and is not defined within it but its binding is not available in the closure of g since it appears in the argument position. In such a situation, all the functions to which f can be applied in the program should be annotated. The solution to this problem is not straightforward since one needs to consider cases such as the following.

```
fun k x = ... f a ...     (* annotate a *)

fun k x = ... f x ...     (* annotate any possible binding of x *)

fun k x = ... x a ...
... k f ...               (* annotate a *)

fun k h y = ... h y ...
... k f ...               (*annotate any possible binding of y *)
```

Example 2

A similar difficulty arises when a higher-order function is transmitted.

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

The function f needs to be annotated. Even if g is defined within f it appears to be a safe option to annotate it as well because it escapes the definition of f. It may well be subsequently transmitted by some other code which receives it.

2.2 Related Work

As is demonstrated by the preceding examples, in languages with higher-order functions, the flow of control from one program point to another is not easily detectable since a function can be passed around and subsequently called from anywhere in the program. A wide range of analyses have been devised to approximate which functions can be called from a particular point in the program. Some instances of these static analyses are also known as closure analysis, set-based analysis and constraint-based analysis which differ in their formulations, the precision they offer and their practicality [9, 10, 11, 12, 13, 14].

More recently, 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 [15] and in extending type systems to perform control flow analyses [16].

It has also been observed by these authors that type inference and control flow analysis are based on different perspectives. Type inference systems reason locally by associating a type with each expression and deriving a type for the program compositionally whereas control flow analyses reason globally and are usually not compositional. 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 [17, 18].

Variations for control flow analyses developed for concurrent programming languages such as [10, 11, 13] could offer a solution to the problem of detecting potentially transmissible functions. However, an effect-based analysis which exploits the existing type system would be more easily applicable. The latter is also the one which is proposed by [3].

We propose an effect-based solution to the above problem which also borrows ideas from other control flow analysis methods. For the problem in hand, it is possible to isolate our point of interest, which is the transmission of functions and integrate a small flow analysis component into the type system rather than performing a full-fledged control flow analysis.

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 and recursive expressions are labelled. Note that all the labels (l) in the program should be distinct. The purpose of labels is to uniquely identify functions.

$$\tau \quad ::= \quad \texttt{Unit} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{chan}(\tau) \mid \tau \rightarrow \tau$$

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

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 to be denoted by primitive operators (predefined 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 and 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.

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

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. The dynamic semantic rules for these are given in section 3.2.3.

$$\frac{CI, E_1 \vdash_{loc_1} e_1 \overset{w_1}{\Longrightarrow} val_1, CI' \quad CI, E_2 \vdash_{loc_2} e_2 \overset{w_2}{\Longrightarrow} val_2, CI' \quad w_1 \parallel w_2 \hookrightarrow \epsilon}{CI, (loc_1[e_1] \parallel loc_2[e_2]) \overset{\epsilon}{\Longrightarrow} (val_1, val_2), CI'}$$

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

$$\epsilon \parallel \epsilon \hookrightarrow \epsilon$$

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

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

3.2.3 Sequential Evaluation Rules

A judgment of the form $CI, E \vdash e \overset{w}{\Longrightarrow} val, CI'$ states that expression e evaluates to value val against an environment E and a global channel identifier set CI updating the channel identifier set to CI'. The evaluation relation $\Longrightarrow$, which is defined in Figure 1, is annotated with possible communication events which may occur during evaluation. E initially 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 accessible and unique.

APPLY serves as a semantic tool to define the application of a predefined operation to an argument.

4 An Effect-Based Type System

The type system specified in the rest of this section is designed with the intention of exposing which functions an expression can possibly transmit. Since our language allows anonymous functions, a name based identification of functions is not appropriate. Instead we use labels. One can think of a label as a particular point in the representation of the program code.

4.1 Semantic Objects

We assume monomorphic types (τ) which classify values, communication effects (κ) which classify communication behaviours we are concerned with, and abstract function closures (v).

Drawing inspiration from previous work by several authors [19, 20, 21] to expose memory allocation or communication behaviours of programs, we annotate types with a combination of effects and abstract function closures.

$$
\begin{array}{lcl}
v & ::= & \emptyset \mid \langle \rho, l \rangle \mid v \cup v \\
\tau & ::= & Unit \mid Int \mid Bool \mid chan_{\chi}(|\tau|) \mid \tau \overset{v,\kappa,v}{\rightarrow} \tau \\
\kappa & ::= & \emptyset \mid out_{\chi}(L) \mid in_{\chi}(\tau) \mid \kappa \cup \kappa
\end{array}
$$

The set v conservatively estimates the set of functions an expression can evaluate to. A function closure is abstracted by $\langle \rho, l \rangle$ where l stands for the label of the function abstraction and ρ contains bindings of free variables of the function expression with functional types. The operator $\cup$ denotes set union.

The set κ estimates the communication effects which may occur during the evaluation of the expression. An expression may have no effect, or its overall effect can be estimated by a set of possible effects; sending a function closure of type τ on a channel whose label is approximated by the set L and receiving a value on a channel.

If an expression has type $\tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2$ it is a function which takes an argument of type τ_1 and gives a result of type τ_2 where v_1 estimates the set of functions which may be passed as an argument and v_2 estimates the set of functions it may return when it is applied. The set κ is the latent effect which may occur when the function is applied. We define $|\tau|$ to be the type resulting from stripping away all the effect(κ) and value(v) annotations from the type τ. Note

(unit) $$CI, E \vdash () \stackrel{\epsilon}{\Longrightarrow} val_{unit}, CI$$

(true) $$CI, E \vdash \mathtt{true} \stackrel{\epsilon}{\Longrightarrow} val_{true}, CI$$

(false) $$CI, E \vdash \mathtt{false} \stackrel{\epsilon}{\Longrightarrow} val_{false}, CI$$

(int) $$CI, E \vdash n \stackrel{\epsilon}{\Longrightarrow} val_n, CI$$

(chan) $$CI, E \vdash \mathtt{channel}(\tau) \stackrel{\epsilon}{\Longrightarrow} ci, CI \cup \{ci\} \quad ci \notin CI$$

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

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

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

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

(primop) $$\frac{CI, E \vdash \mathtt{primop}\, c \stackrel{w_1}{\Longrightarrow} b, CI' \quad CI', E \vdash e \stackrel{w_2}{\Longrightarrow} val, CI'' \quad \mathrm{APPLY}(b, val) = val_1}{CI, E \vdash \mathtt{primop}\, c\, e \stackrel{w}{\Longrightarrow} val_1, CI''}$$

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

(if-t) $$\frac{CI, E \vdash e_1 \stackrel{w_1}{\Longrightarrow} val_{true}, CI' \quad CI', E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val_2, CI''}{CI, E \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \stackrel{w_1 w_2}{\Longrightarrow} val_2, CI''}$$

(if-f) $$\frac{CI, E \vdash e_1 \stackrel{w_1}{\Longrightarrow} val_{false}, CI' \quad CI', E \vdash e_3 \stackrel{w_2}{\Longrightarrow} val_2, CI''}{CI, E \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \stackrel{w_1 w_2}{\Longrightarrow} val_2, CI''}$$

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

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

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

Figure 1: Evaluation Rules

that channels are parameterised over non-annotated types. We want to be able to express, for example, an `Int` $\rightarrow$ `Int` channel which can transmit different `Int` $\rightarrow$ `Int` functions where the κ and v annotations of these functions need not be the same.

The subscripts χ which appear in the types and effects denote channel regions. A more detailed explanation of regions can be found in [22, 21]. The intuitive understanding is that each channel is assigned a unique region which we can use in the static semantics as its identification. Without regions we would not have been able to observe from κ which channels take part in which communication. For example an effect $in_{\chi}(\tau)$ tells us that the channel with region χ is the one which may be engaged in this receive event. Having τ alone would not be sufficient to express this.

4.2 Typing Rules

The static semantics for the language assigns a type, an effect and an abstract value (τ, κ, v) to each expression. This is represented by a judgment of the form $\Gamma \vdash e : \tau, \kappa, v$. We have designed the static semantics so that the κ component of the judgment determines the set of functions to be annotated.

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

Definition 1 (Label, AbsCl)

Assuming that *Label* is a function which when given a function as an argument returns its label we define an abstract closure operation *AbsCl* such that $AbsCl(\Gamma, e) = \langle \rho, l \rangle$ where ρ is the environment Γ restricted to the bindings of free identifiers of e with functional type and $Label(e) = l$.

The typing rules given in Figure 2 require little explanation. The rule for the send expression reveals the essence of the type system. All the information with respect to the function closures, embodied in the v component of the typing judgments is processed by the function named *Trans* to extract the labels of the closures that are potentially transmissible. The definition of the function *Trans* is discussed in more detail in the following section.

The side condition in the application rule is used to ensure that the abstract value computed for the argument is consistent with the abstract value which appears in the annotation of the function type. $\subseteq$ is the subset relation for sets.

4.3 Collecting Labels

Trans collects the labels of functions for which the compiler needs to generate transmissible representations.

(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)	$\Gamma \vdash \texttt{channel}(\tau) : chan_\chi(\tau), \emptyset, \emptyset \quad \tau$ non-annotated type

(var)
$$\frac{\Gamma(x) = (\tau, v)}{\Gamma \vdash x : \tau, \emptyset, v}$$

(fn)
$$\frac{\Gamma[x \mapsto (\tau_1, v_1)] \vdash e : \tau_2, \kappa, v_2}{\Gamma \vdash \texttt{fn}^l\, x \Rightarrow e : \tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2, \emptyset, AbsCl(\Gamma, \texttt{fn}^l\, x \Rightarrow e)}$$

(rec)
$$\frac{\Gamma[f \mapsto (\tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2, AbsCl(\Gamma, \texttt{rec}^l\, f(x) \Rightarrow e))][x \mapsto \tau_1, v_1] \vdash e : \tau_2, \kappa, v_2}{\Gamma \vdash \texttt{rec}^l\, f\,(x) \Rightarrow e : \tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2, \emptyset, AbsCl(\Gamma, \texttt{rec}^l\, f(x) \Rightarrow e)}$$

(app)
$$\frac{\Gamma \vdash e_1 : \tau_1 \overset{v,\kappa,v_2}{\rightarrow} \tau_2, \kappa', v' \quad \Gamma \vdash e_2 : \tau_1, \kappa'', v_1 \quad v_1 \subseteq v}{\Gamma \vdash e_1\, e_2 : \tau_2, \kappa \cup \kappa' \cup \kappa'', v_2}$$

(primop)
$$\frac{\Gamma \vdash \texttt{primop}\, c : \tau_1 \overset{v,\kappa,v_2}{\rightarrow} \tau_2, \emptyset, v' \quad \Gamma \vdash e : \tau_1, \kappa', v_1 \quad v_1 \subseteq v}{\Gamma \vdash \texttt{primop}\, c\, e : \tau_2, \kappa \cup \kappa', v_2}$$

(seq)
$$\frac{\Gamma \vdash e_1 : \tau_1, \kappa, v \quad \Gamma \vdash e_2 : \tau_2, \kappa', v'}{\Gamma \vdash e_1\, ;\, e_2 : \tau_2, \kappa \cup \kappa', v'}$$

(if)
$$\frac{\Gamma \vdash e_1 : Bool, \kappa, \emptyset \quad \Gamma \vdash e_2 : \tau, \kappa', v \quad \Gamma \vdash e_3 : \tau, \kappa'', v'}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \tau, \kappa \cup \kappa' \cup \kappa'', v \cup v'}$$

(let)
$$\frac{\Gamma \vdash e_1 : \tau_1, \kappa, v \quad \Gamma[x \mapsto (\tau_1, v)] \vdash e_2 : \tau_2, \kappa', v'}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2, \kappa \cup \kappa', v'}$$

(receive)
$$\frac{\Gamma \vdash e : chan_\chi(\tau), \kappa, \emptyset \quad \tau = |\tau'|}{\Gamma \vdash e? : \tau, \kappa \cup in_\chi(\tau'), \emptyset}$$

(send)
$$\frac{\Gamma \vdash e_1 : chan_\chi(\tau), \kappa, \emptyset \quad \Gamma \vdash e_2 : \tau', \kappa', v \quad Trans(\tau, v) = L \quad \tau = |\tau'|}{\Gamma \vdash e_1!e_2 : Unit, \kappa \cup \kappa' \cup out_\chi(L), \emptyset}$$

Figure 2: Typing Rules

Definition 2 (Trans)

$Trans(\tau, v) = \emptyset$ where τ is a non-arrow type

$Trans(\tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2, \langle \rho, l \rangle) = \{l\} \cup L_1 \cup L_2$
where $L_1 = \bigcup Trans(\tau_x, v_x)$ for all x such that $\rho(x) = (\tau_x, v_x)$
and $L_2 = Trans(\tau_2, v_2)$

$Trans(\tau, v_1 \cup v_2) = L_1 \cup L_2$ where $L_i = Trans(\tau, v_i)$ for $i = 1, 2$

The first case applies where the value sent is a non-function. These values are of no interest for the information we seek.

The second case applies when the value sent is functional and is approximated by a singleton. In this case, the label of the top level closure is included in L and ρ is examined recursively by $Trans$ to detect functions which are referred to by the top level function whose labels are collected in L_1. The set L_2 is necessary to handle the cases where τ_2 is a function type.

The last case applies when the value sent is approximated by a non-singleton set. In this case $Trans$ examines recursively all the elements of the set and merges their results in one big set.

Example 3

$\texttt{let}\, f = \texttt{fn}^l\, x \;\Rightarrow\; ...x...y...\texttt{in}\, ch!f$

Suppose that the type system assigns f an abstract closure $\langle \rho_f, l_f \rangle$.
The environment ρ_f would contain a binding for y if y is a functional value. The definition of $Trans$ ensures that the label of y is included in L by the virtue of being in L_1 because it examines the ρ_f.

To see how $Trans$ handles more esoteric cases, let us recall Example 1 of section 2.1. The type system would assign to the function `h` an abstract value v which is the union of the abstract closures of those functions which may be passed as an argument to `f`. The $Trans$ function goes through all of these by the application of the last rule.

Example 2 of section 2.1 demonstrates the use of L_2 in the definition of the $Trans$ function. The type system would assign to `g` a type of the form $\tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} (\tau_3 \overset{v_3,\kappa,v_3}{\rightarrow} \tau_4)$. The set v_2 abstracts the function `g` which has the type $\tau_3 \overset{v_3,\kappa,v_3}{\rightarrow} \tau_4$. The set L_2 ensures that `g` is examined for annotation as well.

4.4 Consistency of Dynamic and Static Semantics

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 [23, 24, 25, 20]. Our formulation can be likened to that of [20] 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 [23] in the sense that the formalism of relational semantics is enriched by labelling evaluation rules with event sequences to capture the communication behaviour.

As a semantic tool for the proof we introduce the notion of channel environment which is a finite map from channel identifiers to channel types. E is the evaluation environment as defined in section 3 and Γ is the typing environment as defined in section 4.

$$CE ::= [ci_1 \mapsto chan_{\chi_1}(\tau), \ldots, c_n \mapsto chan_{\chi_n}(\tau)]$$

We also define a semantic relation which establishes the correspondence between objects of the dynamic semantics and the static semantics :

$CE \models val : \tau, v$	the value *val* has the type τ and abstract value v
$CE \models E : \Gamma$	the values of E have the types and abstract values of the corresponding Γ
$CE \models w : \kappa$	the communication events in the sequence w respect the communication effects κ

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

Definition 3 ($\preceq$)

$\epsilon \preceq v$	for any v
$\langle \rho_1, l \rangle \preceq v$	if $\exists \langle \rho_2, l \rangle \in v . \rho_1 \subseteq \rho_2$

Definition 4 ($\models$)

$CE \models val_{unit} : Unit, \epsilon$	
$CE \models val_{true} : Bool, \epsilon$	
$CE \models val_{false} : Bool, \epsilon$	
$CE \models val_n : Int, \epsilon$	
$CE \models ci : chan_\chi(\tau), \epsilon$	if $CE(ci) = chan_\chi(\tau)$ where $\tau = \lvert\tau'\rvert$ for some τ'
$CE \models \langle E, \mathtt{fn}^l\, x \Rightarrow e \rangle : \tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2, v_1$	if $\exists \Gamma . \Gamma \vdash \mathtt{fn}^l\, x \Rightarrow e : \tau_1 \overset{v_1,\kappa,v_2}{\rightarrow} \tau_2, \emptyset, v$ and $v_1 \preceq v$ and $CE \models E : \Gamma$
$CE \models E : \Gamma$	if $Dom(E) \subseteq Dom(\Gamma)$ and $\forall x \in Dom(E). CE \models E(x) : v$ such that $v \preceq \Gamma(x)$
$CE \models \epsilon : \emptyset$	
$CE \models ci?val : in_\chi(\tau')$	if $CE \models ci : chan_\chi(\tau), \epsilon$ and $\exists v. CE \models val : \tau', v$ such that $\tau = \lvert\tau'\rvert$
$CE \models ci!val : out_\chi(L)$	if $CE \models ci : chan_\chi(\tau), \epsilon$ and $CE \models val : \tau', v$ such that $Trans(\tau, v) = L$ and $\tau = \lvert\tau'\rvert$
$CE \models w : \kappa$	if $\forall com \in w. CE \models com : \kappa$

Proposition 1 (Consistency)

Assume $CI, (loc_1[e_1] \mid\mid loc_2[e_2]) \overset{\epsilon}{\Longrightarrow} (val_1, val_2), CI'$. For each $e \in \{e_1, e_2\}$ if $CE \models E : \Gamma$ and $\Gamma \vdash e : \tau, \kappa, v$ and $CI, E \vdash e \overset{w}{\Longrightarrow} val, CI'$ where $Dom(CE) \subseteq CI$ then there exists a CE' such that $CE' \models val : \tau, v_1$ and $v_1 \preceq v$ where $Dom(CE') \subseteq CI'$ and $CE' \models w : \kappa$.

Proof 1

The proof is given in Appendix A 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 $i \in \{0, 1\}$.

5 Extensions and Future Work

5.1 Polymorphism

A polymorphic type system is one of the major attractions of a typical functional programming language. Although polymorphism becomes a more delicate issue in the presence of non-applicative features such as references and communication channels, the literature includes satisfactory treatments of these problems [25, 23, 22, 21]. Extending our type system with polymorphism seems to be a matter of adapting to our setting the previous work by other authors.

As it stands, the monomorphic effect-based type system we have presented not only suggests the main idea for a sound method of detecting transmissible functions but also makes a solid contribution to the static analysis of expressions which make no use of polymorphism. In his thesis, Knabe motivates the problem with examples which have monomorphic types only. With our type system we are able to propose a solution to the question he puts forward.

5.2 An Inference Algorithm

The methodology of annotated type and effect systems consists of :(i) devising a semantics for the typed programming language, (ii) expressing a program analysis by means of an annotated type and effect system, (iii) showing the semantic correctness of the analysis,(iv) developing an inference algorithm and proving it sound and complete, and (v) utilising the information for applications [26].

The work presented so far accomplishes the first three steps. Our next aim is to develop an algorithm which will give the system its practicality.

5.3 Cost Analysis

Developing static analyses to estimate the cost of data movement or network references incurred by the mobility of the programs is the current interest of the author of this text. Programming languages for mobile computation employ different strategies in binding to resources of different natures. Estimating the cost of data movement is analogous to estimating how much of the environment at the sender's site needs to be re-established at the receiver's site as well as how it should be done considering the natures of the resources and the type of binding to them.

Annotating types with abstract function closures as well as communication effects provides us with explicit information about the amount of data movement involved in a communication. For example, a simple modification of effect $out_\chi(L)$ to $out_\chi(v)$ would orient the the type system towards cost analysis rather than collecting labels of potentially transmissible functions since v would contain the information about the part of the environment referred to by the transmitted value.

6 Conclusions

In this paper, we focus on the problem of determining statically whether a function is potentially transmissible at run-time. We base our discussion on a Facile-like language in which mobility is supported by the facility of transmitting functions across different sites. We adapt the methodology of annotated type and effect systems to expose the behaviour of programs with respect to transmission of function closures. Our approach also employs ideas from control flow analysis methods.

Facile is a language which can be considered as a combination of the typed λ-calculus and a core set of constructs for concurrency and distribution based on throughly studied calculi. Therefore, the ideas underlying this work could be applicable to a wide range of programming languages which share similar semantic foundations; particularly in designing rich type systems which could give rise to well-founded optimisations in their compilers.

Acknowledgements

I would like to thank my supervisor Stephen Gilmore for useful discussions and the University of Edinburgh for funding my research.

References

[1] F. Nielson, editor. *ML with Concurrency: Design, Analysis, Implementation, and Application.* Springer-Verlag New York, Inc., 1997.

[2] S.L. Peyton Jones, A.Gordon, and S.Finne. Concurrent haskell. In *Proceedings of POPL'96*, pages 295–308, Florida, Jan 1996.

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

[4] 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.

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

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

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

[8] 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.

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

[10] K.L. Gasser, F.Nielson, and H.R.Nielson. Systematic realisation of control flow analysis for CML. In *Proceedings of ICFP '97.* ACM Press, 1997.

[11] F. Nielson and H.R.Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of POPL'97.* ACM Press, 1997.

[12] S. Jaganathan and S.Weeks. A unified treatment of flow anayisis in higher-order languages. In *Proceedings of POPL'95*, pages 393–407. ACM Press, 1995.

[13] P. Steckler. Detecting local channels in Distributed Poly/ML. Technical Report ECS-LFCS-96-340, LFCS, University of Edinburgh, 1996.

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

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

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

[17] 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.

[18] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proceedings of POPL'95*, pages 367–378. ACM Press, 1995.

[19] P. Jouvelot and D.K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of POPL'90*, pages 303–310. ACM Press, 1990.

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

[21] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of POPL'94*. ACM Press.

[22] 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.

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

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

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

[26] F. Nielson. Annotated type and effect systems. ACM Computing Surveys vol. 28 no.2, 1996. Invited position statement for the Symposium on Models of Programming Languages and Computation.

A Consistency Proof

For brevity, we sketch the proof for some selected cases only. We demonstrate the proof for e_1. The proof for e_2 follows symmetrically. Note that because of the communication rule the proof of e_1 refers to the proof of e_2 and vice versa.

$\boxed{\texttt{case}\ ()}$ Suppose $CI, E \vdash () \overset{w}{\Longrightarrow} val, CI'$. CI' must be equal to CI because there is a unique dynamic semantics rule which might have been applied. That is $CI, E \vdash e \stackrel{\epsilon}{\Longrightarrow} val_{unit}, CI$. By the single relevant static semantics rule we also have $\Gamma \vdash () : Unit, \emptyset, \emptyset$. The definition of $\models$ gives $CE \models val_{unit} : Unit, \epsilon$ and $CE \models \epsilon : \emptyset$. Taking CE as a witness for CE' concludes the proof for this case.

$\boxed{\texttt{case}\ channel(\tau)}$ Suppose $CI, E \vdash channel(\tau) \stackrel{w}{\Longrightarrow} val, CI'$. Reasoning by the uniqueness of the rules like the above case , we know that $w = \epsilon$ and val is equal to ci which is a new, globally unique identifier where $CI' = CI \cup \{ci\}$. We also know that $\Gamma \vdash channel(\tau) : chan_{\chi}(\tau), \emptyset, \emptyset$ where τ is a non-annotated type. By taking CE'= $CE[ci \mapsto chan_{\chi}(\tau)]$ we can conclude that $Dom(CE') \subseteq CI'$ and it follows by the definition of $\models$ that $CE' \models ci : chan_{\chi}(\tau), \epsilon$ and $CE' \models \epsilon : \emptyset$.

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

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

$$\frac{\Gamma[f \ \mapsto \ (\tau_1 \stackrel{v_1,\kappa,v_2}{\rightarrow} \tau_2, AbsCl(\Gamma, \texttt{rec}^l\, f(x) \Rightarrow e))][x \ \mapsto \ \tau_1, v_1] \vdash e : \tau_2, \kappa, v_2}{\Gamma \vdash \texttt{rec}^l\, f\,(x) \Rightarrow e : \tau_1 \stackrel{v_1,\kappa,v_2}{\rightarrow} \tau_2, \emptyset, AbsCl(\Gamma, \texttt{rec}^l\, f(x) \Rightarrow e)}$$

We need to prove that $CE \models val : \tau_1 \stackrel{v_1,\kappa,v_2}{\rightarrow} \tau_2, \langle \rho, l\rangle$ for ρ complying with the definition of $AbsCl$. Note that val is a non-well-founded object.

We closely follow the approach of [25] and view $CE \models val : \tau, v$ as a fixed point equation where $\models$ is its greatest solution. The proof then uses the technique of co-induction for this case. Letting U be the set with elements of the form $CE \times val \times \tau \times v$ and letting $F : P(U) \rightarrow P(U)$ be a monotonic operator so that the semantic relation $\models$ is the fixed point of F we define $Q = Q^{max} \cup \{(val, \tau_1 \stackrel{v_1,\kappa,v_2}{\rightarrow} \tau_2, \langle \rho, l\rangle)\}$ and prove that Q is F-consistent.

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

$$\frac{\begin{array}{l} CI, E \vdash e_1 \stackrel{w_1}{\Longrightarrow} \langle E', \texttt{fn}^l\, x \Rightarrow e\rangle, CI' \quad CI', E \vdash e_2 \stackrel{w_2}{\Longrightarrow} val, CI'' \\ CI'', E'[x \ \mapsto \ val] \vdash e \stackrel{w_3}{\Longrightarrow} val', CI''' \end{array}}{CI, E \vdash e_1\, e_2 \stackrel{w_1 w_2 w_3}{\Longrightarrow} val', CI'''}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \stackrel{v,\kappa,v_2}{\rightarrow} \tau_2, \kappa', v' \quad \Gamma \vdash e_2 : \tau_1, \kappa'', v_1 \quad v_1 \subseteq v}{\Gamma \vdash e_1\, e_2 : \tau_2, \kappa \cup \kappa' \cup \kappa'', v_2}$$

By the induction hypothesis on the premise for e_1 and the assumption $CE \models E : \Gamma$ where $Dom(CE) \subseteq CI$, we have $\exists CE'.CE' \models \langle E', \texttt{fn}^l\, x \ \Rightarrow e\rangle : \tau_1 \stackrel{v,\kappa,v_2}{\rightarrow} \tau_2, \langle \rho, l\rangle$ such that $\langle \rho, l\rangle \preceq v'$ and $CE' \models w_1 : \kappa'$ where $Dom(CE') \subseteq CI'$. For this to hold it must be the case that there exists a Γ' with $CE' \models E' : \Gamma'$ and $\Gamma' \vdash \texttt{fn}^l\, x \ \Rightarrow e : \tau_1 \stackrel{v_1,\kappa,v_2}{\rightarrow} \tau_2, \emptyset, \{\langle \rho_1, l\rangle\}$ such that $\rho \subseteq \rho_1$. The static semantics rule **(fn)** requires that $\Gamma'[x \mapsto \tau_1, v] \vdash e : \tau_2, \kappa, v_2$.

By using the induction hypothesis for e_2 we have $\exists CE''.CE'' \models val : \tau_1, v''$ such that $v'' \preceq v_1$ and $CE'' \models w_2 : \kappa''$ where $Dom(CE'') \subseteq CI''$.

Now, by using the side condition $v_1 \subseteq v$, the definition of $\models$ and the fact that the domain of a channel environment can only expand so that $Dom(CE') \subseteq Dom(CE'')$ we can conclude that $CE'' \models E'' : \Gamma''$ where $E'' = E'[x \mapsto val]$ and $\Gamma'' = \Gamma'[x \mapsto (\tau, v)]$.

By using the induction hypothesis again on $CI'', E'[x \mapsto val] \vdash e \overset{w_3}{\Longrightarrow} val', CI'''$ and $\Gamma'[x \mapsto (\tau_1, v)] \vdash e : \tau_2, \kappa, v_2$ we can conclude that $\exists CE'''.CE''' \models val' : \tau_2, v'''$ such that $v''' \in v_2$ and $Dom(CE''') \subseteq CI'''$ and $CE''' \models w_3 : \kappa$. From $CE' \models w_1 : \kappa', CE'' \models w_2 : \kappa''$ and $CE''' \models w_3 : \kappa$ we also have $CE''' \models w_1 w_2 w_3 : \kappa' \cup \kappa'' \cup \kappa$.

$\boxed{\texttt{case } e?}$ The rules **(receive)** of the dynamic and the static semantics impose that the evaluation and the typing are of the form

$$\frac{CI, E \vdash e \overset{w}{\Longrightarrow} ci, CI'}{CI, E \vdash e? \overset{w.ci?val}{\Longrightarrow} val, CI'} \qquad \frac{\Gamma \vdash e : chan_\chi(\tau), \kappa, \emptyset \quad \tau = |\tau'|}{\Gamma \vdash e? : \tau, \kappa \cup in_\chi(\tau'), \emptyset}$$

We need to prove that $\exists CE'.CE' \models val : \tau', v$ and $CE' \models w.ci?val : \kappa \cup in_\chi(\tau')$ where $v \preceq v'$ for some v' and $Dom(CE') \subseteq CI'$. By the assumption $w_1 \;||\; w_2 \hookrightarrow \epsilon$ of the proposition for consistency we know that there must be a complementary evaluation of a send expression where

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

The case for send expressions proves that one can show $\exists CE''.CE'' \models val : \tau', v$ for some v where $Dom(CE'') \subseteq CI'$. By the induction hypothesis on the premise of the receive expression $\exists CE'.CE' \models ci : chan_\chi(\tau), \epsilon$ and $CE' \models w : \kappa$ where $\tau = |\tau'|$ for some τ'. if we take CE'' as a witness for CE', by the definition of $\models$ it follows that $CE' \models w.ci?val : \kappa \cup in_\chi(\tau')$.

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

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

$$\frac{\Gamma \vdash e_1 : chan_\chi(\tau), \kappa, \emptyset \quad \Gamma \vdash e_2 : \tau', \kappa', v \; Trans(\tau, v) = L \quad \tau = |\tau'|}{\Gamma \vdash e_1!e_2 : Unit, \kappa \cup \kappa' \cup out_\chi(L), \emptyset}$$

We need to prove that $\exists CE'.CE' \models val_{unit} : Unit, \epsilon$ and $CE' \models w_1 w_2 ci!val : \kappa \cup \kappa' \cup out_\chi(L)$ where $\epsilon \preceq \emptyset$ and $Dom(CE') \subseteq CI''$.

By the induction hypothesis there exists a CE'' such that $CE'' \models ci : chan_\chi(\tau), \epsilon$ where $Dom(CE'') \subseteq CI'$ and $CE' \models w_1 : \kappa$. A similar argument applies to the second premise and there exists a CE''' such that $CE''' \models val : \tau, v_1$ such that $v_1 \in v$ where $Dom(CE''') \subseteq CI''$ and $CE''' \models w_2 : \kappa'$.

It is straightforward from the definition of $\models$ that any channel environment CE' satisfies $CE' \models val_{unit} : Unit, \epsilon$. By taking CE''' above as CE' and together with the side condition that $Trans(\tau, v) = L$ and $\tau = |\tau'|$ we can conclude that $CE' \models ci!val : out_\chi(L)$ which in turn allows us to conclude $CE' \models w_1 w_2 ci!val : \kappa \cup \kappa' \cup out_\chi(L)$.