A Static Type System for Detecting Potentially Transmissible Functions

Dilsun Kırlı

Laboratory for Foundations of Computer Science
The University of Edinburgh

Functions, Concurrency and Distribution

- Functional languages have features such as higher-order functions, robust type systems and automatic memory management which make them suitable for writing concise, secure, easy to read, easy to maintain and easy to reuse programs.

- These features can also be desirable in the context of concurrent and distributed programming. Implementations of Concurrent ML, Poly/ML, Facile, Concurrent Haskell provide a support for this idea.

- 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.

Facile

- The original Facile language combines Standard ML with a model of concurrency based on CCS and its higher-order mobile extensions such as CHOCS and the π-calculus.

- Frederick Knabe takes Facile as a departure point and by reusing several parts of its design and implementation arrives at a language particularly suitable for the construction and execution of *mobile agents*.

Functions as Agents

Functions in Facile are first-class values : they can be created at run-time, applied to arguments, passed as arguments to other functions and returned as results. The first-class nature of functions provides a convenient way for representing agents.

```
fun getTimeSec () =
    let val TIME{sec=seconds, ...} = gettimeofday()
    in seconds
    end

fun reportTimeAgent () = send(ansChan, getTimeSec())

val remoteTime = (send(clientChan, reportTimeAgent); receive(ansChan))
```

However, functions in Facile are transmitted as compiled code. This restricts their usage to *homogeneous* architectures.

Dealing with Heterogeneity

- A common method is to adopt a standard transmissible representation and to convert each value into this representation before its transmission by the operation called *marshalling*. Upon receipt, the standard representation is converted into the local representation by the dual operation called *unmarshalling*.

- Facile functions are compiled into a function closure which contains the machine code of the function and the bindings from the definition environment of the function. Functions produced at run-time are just new closures containing old code. This fact allows marshalling of code at *compile-time* leading to a better run-time performance.

- It would be very space inefficient to produce a transmissible representation for *all* functions and to store these as well as the local machine code.

Identifying Transmissible Functions

Knabe's approach is to rely on the programmer to *annotate* potentially transmissible functions so that the compiler generates transmissible code for them. He determines the following criteria to guide the users.

- Any function which is passed directly to the send function is potentially transmissible.

- Any function which is referred to by a potentially transmissible function, and which is not defined within it, is potentially transmissible.

Examples [Knabe, 95]

```
let
    xfun square x = x * x
    xfun cube x = x * square x
in
    send(chan,cube)
end
```

```
let
    xfun cube x =
            let fun square x = x * x
            in x * square x
            end
in
    send(chan,cube)
end
```

Harder Example

```
fun makeCube square =
    xfun cube x = x * square x
in
    send(chan,cube)
end
```

Any possible binding of "square" must be annotated.

Harder Example

```
let
    xfun makeAdder x =
        let xfun add y = x + y
        in add
        end
in
    send(chan,makeAdder)
end
```

What if "add" is subsequently passed to some other code?

Automating Annotations

- 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.

- Analyses known as closure analysis, set-based analysis, constrained-based analysis tackle the problem of approximating the functions called from a particular point in the program.

- The intuitive connection between reasoning about types and control flow has been pointed out by some authors. Research has been carried out in extending control flow systems to perform types analysis and vice versa.

- Type and effect based analysis which exploits the existing type system of the language seems to be more readily applicable.

Syntax

$$
\begin{array}{lcll}
\tau & ::= & \mathtt{Unit} \mid \mathtt{Bool} \mid \mathtt{Int} & \\
 & \mid & \mathtt{chan}(\tau) & \\
 & \mid & \tau \rightarrow \tau & \\
 & & & \\
e & ::= & () \mid \mathtt{true} \mid \mathtt{false} \mid n & \textsf{\textit{const}} \\
 & \mid & \mathtt{channel}(\tau) & \textsf{\textit{chan}} \\
 & \mid & \mathtt{primop}\, c\, e & \textsf{\textit{primop}} \\
 & \mid & x & \textsf{\textit{var}} \\
 & \mid & \mathtt{fn}^{l}\, x \Rightarrow e & \textsf{\textit{fn}} \\
 & \mid & e_1\, e_2 & \textsf{\textit{app}} \\
 & \mid & e_1; e_2 & \textsf{\textit{seq}} \\
 & \mid & \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 & \textsf{\textit{let}} \\
 & \mid & \mathtt{rec}^{l}\, f(x) \Rightarrow e & \textsf{\textit{rec}} \\
 & \mid & \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 & \textsf{\textit{cond}} \\
 & \mid & e_1? & \textsf{\textit{receive}} \\
 & \mid & e_1!e_2 & \textsf{\textit{send}}
\end{array}
$$

System Rule

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

Matching

$$\epsilon \;||\; \epsilon \hookrightarrow \epsilon$$

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

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

Selected Rules from the Dynamic Semantics

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

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

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

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

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

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

Static Semantics Objects

We have monomorphic types (τ) which classify values, communication effects (κ) which classify communication behaviours, and abstract function closures (v).

$$
\begin{array}{lll}
v & ::= & \emptyset \mid \langle \rho, l \rangle \mid v \cup v \\
\tau & ::= & Unit \mid Int \mid Bool \mid chan(|\tau|) \mid \tau \stackrel{v,\kappa,v}{\rightarrow} \tau \\
\kappa & ::= & \emptyset \mid out(L) \mid in(\tau) \mid \kappa \cup \kappa
\end{array}
$$

An expression of type $\tau_1 \stackrel{v_1,\kappa,v_2}{\rightarrow} \tau_2$ is a function 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.

Selected Rules from the Static Semantics

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

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

(fn)
$$\frac{\Gamma[x \mapsto (\tau_1, \upsilon_1)] \vdash e : \tau_2, \kappa, \upsilon_2}{\Gamma \vdash \texttt{fn}^l\, x \Rightarrow e : \tau_1 \overset{\upsilon_1, \kappa, \upsilon_2}{\rightarrow} \tau_2, \emptyset, AbsCl(\Gamma, \texttt{fn}^l\, x \Rightarrow e)}$$

(app)
$$\frac{\Gamma \vdash e_1 : \tau_1 \overset{\upsilon, \kappa, \upsilon_2}{\rightarrow} \tau_2, \kappa', \upsilon' \quad \Gamma \vdash e_2 : \tau_1, \kappa'', \upsilon_1 \quad \upsilon_1 \subseteq \upsilon}{\Gamma \vdash e_1\, e_2 : \tau_2, \kappa \cup \kappa' \cup \kappa'', \upsilon_2}$$

(send)
$$\frac{\Gamma \vdash e_1 : chan(\tau), \kappa, \emptyset \quad \Gamma \vdash e_2 : \tau', \kappa', \upsilon \quad Trans(\tau', \upsilon) = L \quad \tau = |\tau'|}{\Gamma \vdash e_1 ! e_2 : Unit, \kappa \cup \kappa' \cup out(L), \emptyset}$$

Forming an Abstract Closure

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 $Label(e) = l$.

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

The type system assigns f an abstract closure $\langle \rho_f, l_f \rangle$ so that the environment ρ_f contains a binding for y if y is a functional value.

Collecting Labels

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

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 $i = 1, 2$

Consistency of Static and Dynamic Semantics

We 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

Proposition (Consistency)

Assume $CI, (loc_1[e_1] \ || \ loc_2[e_2]) \stackrel{\epsilon}{\Longrightarrow} (val_1, val_2), CI'$. For each $e \in \{e_1, e_2\}$ if $CE \models E : \Gamma$ and $\Gamma \vdash e : \tau, \kappa, v$ and $CI, E \vdash e \stackrel{w}{\Longrightarrow} val, CI'$ 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 $CE' \models w : \kappa$.

Future Work

- Polymorphism
- A Type Inference Algorithm
- Cost Analysis

Conclusions

- The methodology of annotated type and effect systems can be adapted to expose the behaviour of programs with respect to transmission of function closures.

- Since a function closure contains bindings of the free identifiers of the function, obtaining information about the transmissibility of functions also provides information about the data movement behaviour of programs.

- The main idea of this work can be applicable to a wide range of programming languages which have similar semantics foundations as that of Facile.

Annotated Example

fun makeCube square =
    xfun cube x = x * square x
in send(chan,cube)
end

cube : $int \stackrel{\emptyset,\emptyset,\emptyset}{\rightarrow} int, \emptyset, \langle \rho_{cube}, l_{cube} \rangle$ ρ_{cube} includes $\langle \rho_a, l_a \rangle \cup v_2$

square :$int \stackrel{\emptyset,\emptyset,\emptyset}{\rightarrow} int, \kappa, \langle \rho_a, l_a \rangle \cup v_2$

makeCube :$(int \stackrel{\emptyset,\emptyset,\emptyset}{\rightarrow} int) \stackrel{\langle \rho_a, l_a \rangle \cup v_2, \kappa', \emptyset}{\rightarrow} unit, out(L), \langle \rho, l_{makeCube} \rangle$

$$L = \{l_{cube}\} \cup Trans(int \stackrel{\emptyset,\emptyset,\emptyset}{\rightarrow} int, \langle \rho_a, l_a \rangle) \cup Trans(v_2)$$

Annotated Example

let
  xfun makeAdder x $=$
    let xfun add y $=$ x $+$ y
    in add
    end
in
  send(chan,makeAdder)
end : $unit, out(L), \emptyset$

makeAdder : $int \xrightarrow{\emptyset,\emptyset,\langle\emptyset,l_{makeAdder}\rangle} (int \xrightarrow{\emptyset,,\emptyset} int), \emptyset, \langle\emptyset, l_{makeAdder}\rangle$

add :$int \xrightarrow{\emptyset,,\emptyset} int, \emptyset, \langle\emptyset, l_{add}\rangle$

$$L = \{l_{makeAdder}\} \cup Trans(int, \emptyset) \cup Trans(\emptyset, \langle\emptyset, l_{add}\rangle)$$

```
let
    xfun square x = x * x
    xfun cube x = x * square x
in
    send(chan,cube)
end
```

cube : $int \overset{\emptyset,\emptyset,\emptyset}{\rightarrow} int, \emptyset, \langle \rho_{cube}, l_{cube} \rangle$
ρ_{cube} includes the binding for v_{square}
$L = \{l_{cube}\} \cup \{l_{square}\}$

```
let
    xfun cube x =
        let fun square x = x * x
        in x * square x
        end
in
    send(chan,cube)
end
```

cube : $int \overset{\emptyset,\emptyset,\emptyset}{\rightarrow} int, \emptyset, \langle \rho_{cube}, l_{cube} \rangle$
ρ_{cube} is empty
$L = \{l_{cube}\}$