The RPC Calculus
Symmetrical RPC in an Asymmetrical World

Ezra Cooper       Philip Wadler

September 8, 2009
Dream of unified web programming
Dream of unified web programming

What we want
Reality of web programming
Reality of web programming

It’s a bit more fiddly.
Reality of web programming

It’s a bit more fiddly.

Here’s why:
Traditional program
Traditional program

User

Program

Disk
Traditional web program
Traditional web program
Unified program

Unified Web Program

http://groups.inf.ed.ac.uk/links
Unified program

Simplifies the work of web programming

User | Unified Web Program | Disk

Client | Network | Server Program | Disk
Unified program

Unified Web Program

Simplifies the work of web programming

The Links language:

http://groups.inf.ed.ac.uk/links
Dream of a unified language

Waking up:

- Desire to control location explicitly, with a light touch;
- Need control for performance and security reasons;
- Tricky because of asymmetrical client/server relationship.
Roadblock: Asymmetrical client/server relationship
Roadblock: Asymmetrical client/server relationship
Stateless server

Web applications should not store control state at the server.
Stateless server

Web applications should not store control state at the server.

Server should encode all state and give it to client.
Stateless server

Web applications should not store control state at the server.

Server should encode all state and give it to client.

For this talk, \textit{state} = call stack.
Example program

```kotlin
fun checkPassword(name, password) {
    // load this user’s row from database & check password
    var u = lookupUser(name);
    u.password == password
}

fun validate() {
    var auth = checkPassword(fieldValue("name"),
                              fieldValue("password"));
    if (auth)
        displaySecretDocument();
    else
        displayErrorMessage();
}
```
fun checkPassword(name, password) server {
    # load this user’s row from database & check password
    var u = lookupUser(name);
    u.password == password
}

fun validate() client {
    var auth = checkPassword(fieldValue("name"),
                              fieldValue("password"));

    if (auth)
        displaySecretDocument();
    else
        displayErrorMessage();
}
fun findFlights(flightQuery) server {
    # Query each vendor for its own matching flights
    for (vendor ← airlines()) {
        var flights = queryVendor(vendor, flightQuery);
        // Send this vendor’s flights to the browser
        displayFlights(flights);
    }
}

fun displayFlights(flights) client {
    // Add each flight to the page
    for (flight ← flights)
        addToPage(flight);
}
Example: higher-order functions

How should this code behave?

```javascript
fun usernameMap(f) server {
  var users = getUsersFromDatabase();
  for (u ← users)[f(u.name)]
}

fun userNameFirstThree() client {
  usersMap(fun(name){take(3, name)});
}
```
Example: higher-order functions

How should this code behave?

```plaintext
fun usernameMap(f) server {
    var users = getUsersFromDatabase();
    for (u ← users)[f(u.name)]
}

fun userNameFirstThree() client {
    usersMap(fun(name){take(3, name)});
}
```

▶ Functions in *lexical* client-context execute on client.
What I want to show you

- How to compile this language for the asymmetrical client-server model,
What I want to show you

- How to compile this language for the asymmetrical client-server model,
- How the compilation factors into standard techniques,
What I want to show you

- How to compile this language for the asymmetrical client-server model,
- How the compilation factors into standard techniques,
- How these techniques can be presented formally and concisely.
How it’s done

Call to f (server)
Call to g (client)
Return r from g
Return s from f

main Client Server
Source language:
call/return style
Implementation:
request/response style

Call to f (server)
Call to g (client)
Return r from g
Return s from f

Client

Server

{Call f}
{Call g, k}
{Continue r, k}
{Return s}
Getting technical
Source language: the located lambda calculus
Source language: the located lambda calculus

\[ L, M, N ::= LM \mid \lambda^a x. N \mid \lambda x. N \mid x \mid c \]
\[ a, b ::= c \mid s \]
Source language: the located lambda calculus

\[ L, M, N \ ::=  \; LM \mid \lambda^a x. N \mid \lambda x. N \mid x \mid c \]

\[ a, b \ ::=  \; c \mid s \]

We eliminate un-located forms \( \lambda x. N \) by explicitly copying the location of their lexical context.

So \( \lambda^c x. L(\lambda y. N) \) becomes \( \lambda^c x. L(\lambda^c y. N) \)
Source language: the located lambda calculus

\[ L, M, N ::= LM | \lambda^a x. N | x | c \]
\[ a, b ::= c | s \]
Semantics

Read $M \downarrow_a V$ as "$M$ evaluates, starting at $a$, to $V$.”

\[
V \downarrow_a V \quad \textit{(VALUE)}
\]

\[
\frac{L \downarrow_a \lambda^b x. N \quad M \downarrow_a W \quad N\{W/x\} \downarrow_b V}{LM \downarrow_a V} \quad \textit{(BETA)}
\]

\[
\frac{L \downarrow_a c \quad M \downarrow_a W \quad \delta_a(c, W) \downarrow_a V}{LM \downarrow_a V} \quad \textit{(DELTA)}
\]
Translation to a client-server system

Three techniques:

- **CPS translation:**
  reifies the control state

- **Defunctionalization:**
  turns higher-order functions into data (serializable)

- **Trampolining:**
  inverts control, so state resides at client.
CPS translation
(due to Fischer, 1972, via Sabry and Wadler, 1997)

Source:

\[ L, M, N ::= LM | V \]
\[ V ::= \lambda x. N | x \]

CPS translation:

\[
(LM)^\dagger K = L^\dagger (\lambda f. M^\dagger (\lambda x. fxK)) \]
\[ V^\dagger K = KV^\circ \]

\[
(\lambda x. N)^\circ = \lambda x. \lambda k. N^\dagger k \]
\[ x^\circ = x \]
Defunctionalization
Defunctionalization target

\[ D ::= \texttt{letrec} \ D \ 	extbf{and} \ \cdots \ 	extbf{and} \ D \]

\[ D ::= f(\vec{x}) = \texttt{case} \ x \ 	extbf{of} \ \mathcal{A} \]

\[ \mathcal{A} ::= \text{a set of } \mathcal{A} \text{ items} \]

\[ \mathcal{A} ::= F(\vec{c}) \Rightarrow M \]

\[ M ::= f(\vec{M}) | F(\vec{M}) | x | c \]
Defunctionalization target

\[ D ::= \text{letrec } D \text{ and } \cdots \text{ and } D \]
\[ D ::= f(\vec{x}) = \text{case } x \text{ of } A \]
\[ A ::= \text{a set of } A \text{ items} \]
\[ A ::= F(\vec{c}) \Rightarrow M \]
\[ M ::= f(\vec{M}) | F(\vec{M}) | x | c \]

function names  \( f, g \)
constructor names  \( F, G \)
Defunctionalization (orig. Reynolds, 1972)

\[
[[M]]^{\text{top}} = \text{letrec } apply(func, arg) = \text{case } func \text{ of } [[M]]^* \\
\text{in } [[M]]
\]

\[
[[\lambda x. N]]^{\text{fun}} = \Gamma \lambda x. N \uparrow (\vec{y}) \Rightarrow [[N]]\{arg/\chi\}
\]

where \( \vec{y} = \text{FV}(\lambda x. N) \)

\[
[[LM]] = apply([[L]], [[M]])
\]

\[
[[V]] = V^\circ
\]

\[
(\lambda x. N)^\circ = \Gamma \lambda x. N \uparrow (\vec{y})
\]

where \( \vec{y} = \text{FV}(\lambda x. N) \)

\[
\chi^\circ = \chi
\]

The operation \( \Gamma M \uparrow \) gives an opaque identifier for the term \( M \).
Trampolining (due to Ganz, Friedman and Wand)

- Continually returns control to a top-level trampoline;
- Works on any tail-form program, including CPS programs;
- Choice of the trampoline modifies the behavior.
Trampolining

\[(LM)^T = \text{Bounce}(\lambda z. L^t M^t)\]

(\text{where } z \text{ is a dummy})

\[V^T = \text{Return}(V^t)\]

\[(\lambda x. N)^t = \lambda x. N^T\]

\[x^t = x\]

Behavior depends on our choice of \textit{tramp}. 
Trivial trampoline:

\[
tramp(x) = \text{case } x \text{ of}
\]

\[
Bounce(thunk) \Rightarrow tramp(thunk())
\]

\[
| \text{Return}(x) \Rightarrow x
\]
Example trampolines

Trivial trampoline:

\[
tramp(x) = \text{case } x \text{ of} \\
\quad \text{Bounce(thunk)} \Rightarrow \text{tramp(thunk())} \\
\quad | \text{Return}(x) \Rightarrow x
\]

Step-counting trampoline:

\[
tramp(n, x) = \text{case } x \text{ of} \\
\quad \text{Bounce(thunk)} \Rightarrow \text{print}(n); \text{tramp}(n + 1, \text{thunk()}) \\
\quad | \text{Return}(x) \Rightarrow x
\]
Our trampoline

Since the control state is reified, tramp can split the computation into a client- and a server-side piece.

\[
tramp(x) = \text{case } x \text{ of } \\
| Bounce(f, x, k) \Rightarrow \\
\quad tramp(\text{req cont}(k, \text{apply}(f, x))) \\
| Return(x) \Rightarrow x
\]

(This shouldn’t make sense yet; don’t worry.)
Our trampoline

Since the control state is reified, *tramp* can split the computation into a client and a server-side piece.

\[
\text{tramp}(x) = \text{case } x \text{ of} \\
\quad | \ Call(f, x, k) \Rightarrow \\
\quad \quad \quad \quad \text{tramp}(\text{req cont}(k, \text{apply}(f, x))) \\
\quad | \ Return(x) \Rightarrow x
\]

(This shouldn’t make sense yet; don’t worry.)
The Big Transformation
First, the target: first-order client-server calculus
The client-server calculus

Syntax

configurations $\mathcal{K} ::= (M; \cdot) \mid (E; M)$

terms $L, M, N ::= x \mid c \mid F(\vec{M}) \mid f(\vec{M}) \mid \text{req } f(\vec{M})$

definition set $\mathcal{D}, \mathcal{C}, \mathcal{S} ::= \text{letrec } D \text{ and } \cdots \text{ and } D$

function definitions $D ::= f(\vec{x}) = \text{case } M \text{ of } A$

alternative sets $A ::= \text{a set of } A \text{ items}$

case alternatives $A ::= F(\vec{x}) \Rightarrow M$

function names $f, g$

constructor names $F, G$
Configurations of the machine

Active client

Idle server

Waiting client

Active server

$(M; \cdot )$

$(E; M)$
Semantics

Client:

\[(E[f(\vec{V})]; \cdot) \rightarrow_{c,s} (E[M\{\vec{V}/\vec{x}\}]; \cdot) \quad \text{if } (f(\vec{x}) = M) \in C\]

\[(E[\text{case } (F(\vec{V})) \text{ of } \mathcal{A}]; \cdot) \rightarrow_{c,s} (E[M\{\vec{V}/\vec{x}\}]; \cdot) \quad \text{if } (F(\vec{x}) \Rightarrow M) \in \mathcal{A}\]

Server:

\[(E; E'[f(\vec{V})]) \rightarrow_{c,s} (E; E'[M\{\vec{V}/\vec{x}\}]) \quad \text{if } (f(\vec{x}) = M) \in S\]

\[(E; E'[\text{case } (F(\vec{V})) \text{ of } \mathcal{A}]) \rightarrow_{c,s} (E; E'[M\{\vec{V}/\vec{x}\}]) \quad \text{if } (F(\vec{x}) \Rightarrow M) \in \mathcal{A}\]

Communication:

\[(E[\text{req } f (\vec{V})]; \cdot) \rightarrow_{c,s} (E; f(\vec{V}))\]

\[(E; V) \rightarrow_{c,s} (E[V]; \cdot)\]
Now, the translation
Transformation on terms

\[(\lambda^a x. N)^\circ = \neg\lambda^a x. N\neg(\vec{y})\]
\[x^\circ = x\]
\[c^\circ = c\]

\[V^* = V^\circ\]
\[(LM)^* = apply(L^*, M^*)\]

\[V^\dagger[ ] = cont([ ], V^\circ)\]
\[(LM)^\dagger[ ] = L^\dagger(\neg M\neg(\vec{y}, [ ]))\]
where \(\vec{y} = FV(M)\)
Transformation to definitions (client-side)

\[
[M]_{c, \text{top}} = \text{letrec } apply(\text{fun}, \text{arg}) = \text{case fun of } [M]_{c, \text{fun}} \\
\text{and tramp}(x) = \text{case } x \text{ of } \\
| \text{Call}(f, x, k) \Rightarrow \\
\quad \text{tramp(req cont } (k, \text{apply}(f, x))) \\
| \text{Return}(x) \Rightarrow x
\]

\[
[\lambda^c x. N]_{c, \text{fun}} = \Gamma \lambda^c x. N \downarrow (\vec{y}) \Rightarrow N^* \{\text{arg} / x\} \\
\text{where } \vec{y} = \text{FV}(\lambda x. N)
\]

\[
[\lambda^s x. N]_{c, \text{fun}} = \\
\Gamma \lambda^s x. N \downarrow (\vec{y}) \Rightarrow \text{tramp(req apply } (\Gamma \lambda^s x. N \downarrow (\vec{y}), \text{arg}, \text{Fin}())\text{)} \\
\text{where } \vec{y} = \text{FV}(\lambda x. N)
\]
Transformation to definitions (server-side)

\[
\begin{align*}
\llbracket M \rrbracket^{s,\text{top}} &= \text{letrec } \text{apply}(\text{fun}, \text{arg}, k) = \text{case } \text{fun} \text{ of } \llbracket M \rrbracket^{s,\text{fun}} \\
\text{and } \text{cont}(k, \text{arg}) &= \text{case } k \text{ of } \\
\quad &\llbracket M \rrbracket^{s,\text{cont}} \\
\quad | \text{App}(\text{fun}, k) \Rightarrow \text{apply}(\text{fun}, \text{arg}, k) \\
\quad | \text{Fin()} \Rightarrow \text{Return}(\text{arg}) \\
\llbracket \lambda^s x. N \rrbracket^{s,\text{fun}} &= \overset{\text{where } \vec{y} = \text{FV}(\lambda x. N)}{\neg \lambda^s x. N^{-}(\vec{y}) \Rightarrow (N^\dagger k)\{\text{arg} / x\}} \\
\llbracket \lambda^c x. N \rrbracket^{s,\text{fun}} &= \overset{\text{where } \vec{y} = \text{FV}(\lambda x. N)}{\neg \lambda^c x. N^{-}(\vec{y}) \Rightarrow \text{Call}(\neg \lambda^c x. N^{-}(\vec{y}), \text{arg}, k)} \\
\llbracket LM \rrbracket^{s,\text{cont}} &= \overset{\text{where } \vec{y} = \text{FV}(M)}{\neg M^{-}(\vec{y}, k) \Rightarrow M^\dagger(\text{App}(\text{arg}, k))}
\end{align*}
\]
Correctness: Bisimulation
Summary

We can
Summary

We can

- Enrich a functional programming language with *location annotations*,

Summary

We can

- Enrich a functional programming language with \textit{location annotations},
- which designate execution location of their contents lexically,
We can

- Enrich a functional programming language with *location annotations*,
- which designate execution location of their contents lexically,
- and whose semantics are straightforward,
We can

- Enrich a functional programming language with *location annotations*,
- which designate execution location of their contents lexically,
- and whose semantics are straightforward,
- and we can execute these programs on an asymmetrical client-server system with a “stateless server”
Summary

We can

▶ Enrich a functional programming language with location annotations,
▶ which designate execution location of their contents lexically,
▶ and whose semantics are straightforward,
▶ and we can execute these programs on an asymmetrical client-server system with a “stateless server”
▶ using a combination of classic transformations.
Summary

We can

- Enrich a functional programming language with *location annotations*,
- which designate execution location of their contents lexically,
- and whose semantics are straightforward,
- and we can execute these programs on an asymmetrical client-server system with a “stateless server”
- using a combination of classic transformations.

Thank you