Programs + Communication Structures = Systems

Simon Gay
University of Glasgow
Introduce a new slogan:

programs + communication structures = systems

Structured communication via session types.

Current projects.
Programs + Communication Structures = Systems
For a long time (1950s – 1990s), most computing consisted of isolated computers doing data processing.

The importance of structured data was realised very early. The first high-level programming languages supported data structures and data types.
Data structure declarations in Cobol:

```
*---------------------------------------------------------------*
*     DATA-NAME                  DATA-TYPE                  *
*---------------------------------------------------------------*
  01  PRINCIPAL  PIC 9999.
  01  NUMBER-OF-YEARS  PIC 99.
  01  RATE-OF-INTEREST  PIC 99.
```
Data structure declarations in Fortran 77:

```
INTEGER   COLS, ROWS
PARAMETER(ROWS=12, COLS=10)
REAL      MATRIX(ROWS, COLS), VECTOR(ROWS)
```
Lisp uses (dynamically typed) lists as a universal data structure:

```
(cons '(1 2) '(3 4))
;Output: ((1 2) 3 4)
```
Niklaus Wirth, inventor of the programming language Pascal, introduced the slogan “algorithms + data structures = programs”.
Programming languages allow data structures to be codified as data types. Programming tools and environments use data types as the basis for analysis and verification:

- at compile time, in languages such as Java, C#, Scala, Haskell
- at run time, in languages such as Python

Example, when programming in Java with Eclipse:

- a red X if you apply an operation to the wrong data type
- a menu suggesting appropriate operations for a data type
Computing has changed. We now depend on systems of communicating programs:

- web applications and web services
- mobile apps and their connections to servers
- cloud computing
- data centres

Even within a single computer, further speed increases will depend on communicating many-core programs.
I propose a new slogan for the era of communication:

\[
\text{programs + communication structures = systems}
\]

Communication structures are already an essential part of the design of systems, but we need to codify them as types so that programming languages and environments can use them.

The analogue of the data type, for the era of communication, is the session type [Honda 1993; Takeuchi, Honda & Kubo 1994; Honda, Vasconcelos & Kubo 1998].
Structured communication via session types
Type-theoretic specification of communication protocols, so that protocol implementations can be verified by static type-checking.

Maths server protocol (server side)

\[ T = \{ \text{plus: } ?\text{int.}?\text{int.}!\text{int.}T, \]
\[ \quad \text{neg: } ?\text{int.}!\text{int.}T, \]
\[ \quad \text{quit: } \text{end} \} \]

Maths server protocol (client side)

\[ S = \{ \text{plus: } !\text{int.}!\text{int.}?\text{int.}S, \]
\[ \quad \text{neg: } !\text{int.}?\text{int.}S, \]
\[ \quad \text{quit: } \text{end} \} \]

\[ S = \text{dual}(T) \]
Assume that we are working in a concurrent or distributed system, with point-to-point communication channels (like pi-calculus).

Channels are bi-directional (in practice they may be implemented by pairs of uni-directional channels).

Communication may be synchronous (i.e. sender and receiver both block), or asynchronous with message queues (only receiver blocks).
Session Types in Detail: Maths Server

T = &{ plus: ?int.?int.!int.T,
eg: ?int.!int.T,
quit: end }

offer a choice

receive

send

three options

recursion
Session Types in Detail: Maths Client

\[ S = \oplus \{ \text{plus: } !\text{int}!\text{int}?\text{int}.S, \]
\[ \text{neg: } !\text{int}?\text{int}.S, \]
\[ \text{quit: end } \} \]
S = \{ plus: !int.!int.?int.S, 
    neg: !int.?int.S, 
    quit: end \}

request connection c : S from maths.org:75
select plus on c
send 2 on c
send 3 on c
receive x from c
send quit on c
compute with x
S = \{ \text{plus: } !\text{int} \cdot !\text{int} \cdot ?\text{int} \cdot S, \\
\text{neg: } !\text{int} \cdot ?\text{int} \cdot S, \\
\text{quit: end} \} \\

request connection c : S from maths.org:75 \\
select plus on c \\
send 2 on c \\
send 3 on c \\
receive x from c \\
compute with x \\
send quit on c \\
assume a trusted registry of typed services \\
only select is allowed \\
only send is allowed \\
only receive is allowed \\
after this point, c cannot be used \\
x must be used as int \\
must be int \\
must be plus, neg or quit
\[ S = \oplus \{ \text{plus: } !\text{int}!\text{int}?.\text{int}.S, \]
\[ \text{neg: } !\text{int}?.\text{int}.S, \]
\[ \text{quit: } \text{end} \} \]

Request connection \( c : S \) from maths.org:75

Select plus on \( c \)

Send 2 on \( c \)

Send 3 on \( c \)

Receive \( x \) from \( c \)

Send quit on \( c \)

Compute with \( x \)
T = \{ plus: \text{?int.?int.!int.T,} \\
\quad \text{neg: \text{?int.!int.T,}} \\
\quad \text{quit: \text{end}} \}

accept connection c : T on port 75
label start:
\quad offer on c {
\quad \quad plus: receive x from c \\
\quad \quad receive y from c \\
\quad \quad send x+y on c \\
\quad \quad goto start \\
\quad \quad neg: receive z from c \\
\quad \quad send –z on c \\
\quad \quad goto start \\
\quad \quad quit: break 
\}
Static Typechecking: Maths Server

T = &{ plus: ?int.?int.!int.T,
    neg: ?int.!int.T,
    quit: end }

accept connection c : T on port 75

label start:

offer on c {
    plus: receive x from c
            receive y from c
            send x+y on c
            goto start

    neg: receive z from c
            send –z on c
            goto start

    quit: break
}

only offer is allowed

all options must be present

looping is only allowed when the type
of c has returned to its initial state

“goto statement considered typeful”
It is essential that each endpoint of a channel is used by only one component of a system.

one select would go first, then the other would be incorrect: race condition
To guarantee unique ownership of channel endpoints, session type systems use standard techniques of linearity [Girard 1987].

Specific techniques for linear type systems may be based on e.g. [Kobayashi, Pierce & Turner 1996] for pi-calculus, or [Wadler 1990] for functional languages.

Unique ownership is also guaranteed in the presence of delegation, i.e. sending a session-typed channel in a message.
The POP3 Protocol

START +OK string → AUTHORIZATION

AUTHORIZATION

QUIT
+OK string

END

USER string
+OK string

PASS string
+OK string

QUIT
+OK string

END

TRANSACTION

QUIT

+OK string

END

TRANSACTION

QUIT

+OK int x int

STAT

+OK string string

client message

server message

+OK string string

-ERR string
S = START, A = AUTHORIZATION, T = TRANSACTION

S = ⊕{ok : !Str . A}

A = &{quit : ⊕{ok : !Str . end},
    user : ?Str . ⊕{error : !Str . A,
        ok : !Str . &{quit : ⊕{ok : !Str . end},
            pass : ?Str . ⊕{error : !Str . A, ok : !Str . T}}}}

T = &{stat: ⊕{ok: !(Int×Int).T},
    retr: ?Int.⊕{ok: !Str.!Str.T,
        error: !Str.T},
    quit : ⊕{ok : !Str . end}}
Session types in programming languages
\[ T = \{ \text{plus: } \texttt{?int.\texttt{?int.\texttt{!int.\texttt{T}}},} \\
\quad \text{neg: } \texttt{?int.\texttt{!int.\texttt{T}}},} \\
\quad \text{quit: end} \} \]

\[ a : [ T ] \text{ is a typed access point (a “well known name”) } \]

\[ \text{server = accept c on a . loop} \]
\[ \text{loop = c} \rightarrow\{ \text{plus: } c[\texttt{x}] . c[\texttt{y}] . c[\texttt{x+y}] . \text{loop},} \\
\quad \text{neg: } c[\texttt{z}] . c[\texttt{-z}] . \text{loop},} \\
\quad \text{quit: 0} \} \]
Assume an infrastructure for linear typing, and add session types.

\[
\begin{align*}
\text{accept} : [S] & \Rightarrow S \\
\text{send} : A & \Rightarrow !A \times S \\
\text{receive} : ?A \times S & \Rightarrow A \\
\text{close} : \text{end} & \Rightarrow \text{unit}
\end{align*}
\]

fun loop c = case c of {
  plus : let (x,c) = receive c in
         let (y,c) = receive c in
         let c = send (x+y) c in
         loop c ,
  neg : let (x,c) = receive c in
       let c = send \(-x\) c in
       loop c ,
  quit : close c }

fun server x = loop (accept x)

[Gay & Vasconcelos 2010]
Encode session types in the existing type system, without linear types; enforce unique ownership by managing channels through an API.

[Neubauer & Thiemann 2004] in Haskell

[Pucella & Tov 2008] in Haskell, OCaml, SML, Java, C#

+ No need to change the language implementation.
+ Type inference for free.

- Type error messages are difficult to understand.
SJ (Session Java) [Hu, Yoshida & Honda 2008] adds session types to Java. The language is implemented and has been applied to scientific computing (parallel algorithms) and event-driven programming.

The session type constructors have been adapted for Java.

A very nice runtime system supports delegation of sessions, and gives the option of socket-based channels or in-memory channels.

The main limitation is that session channels cannot be stored in fields.
class Maths {
    chan{S} c;
    void init() {
        c = request(maths);
    }
    int add(int x, int y) {
        c.send(plus);
        c.send(x);
        c.send(y);
        return c.receive();
    }
    int negate(int x) {
        c.send(neg);
        c.send(x);
        return c.receive();
    }
    int stop() {
        c.send(quit);
    }
}

S = ⊕{ plus: !int.!int.?int.S,
       neg: !int.?int.S,
       quit: end }

Maths m = new Maths();
m.init();
println(m.add(3,4));
println(m.negate(2));
m.stop();
An object of class Maths has non-uniform method availability. It is a non-uniform object.

Initially, only init is available.

After calling init, add and negate can be called repeatedly.

stop is always available after init has been called, but after calling stop, nothing is available.

The study of static type systems for non-uniform objects is called typestate.

We have developed an integration of session types and typestate [Gay, Vasconcelos, Ravara, Gesbert & Caldeira 2010].
Another approach is the “amalgamation” of sessions and objects [Capecchi, Coppo, Dezani, Drossopoulou & Giachino 2009].

An object has sessions instead of methods.

Method call/return is generalised to a longer interaction between client code and an object. It could be distributed, like RMI, or local.

As far as I know, this idea has not been implemented yet.
Multi-party session types
Current projects
Behavioural Types for Reliable Large-Scale Software Systems

Funded by COST (Cooperation on Science and Technology), 4 years from October 2012; €130k per year.

22 countries; approximately 100 people.

Workshops: BEAT at POPL (Jan 2013, Rome)  
BEAT 2 at SEFM (Sep 2013, Madrid).

Summer school in 2014.

Funding for Short-Term Scientific Missions (i.e. visits).

www.behavioural-types.eu
From Data Types to Session Types:
A Basis for Concurrency and Distribution

Phil Wadler (Edinburgh), Nobuko Yoshida (Imperial), SG (Glasgow)

Funded by UK EPSRC for 5 years from June 2013; £3.9m

5 post-doc positions from 1st June (1 @ ED, 2 @ IC, 2 @ GL)
2 PhD studentships at each site (1 from Oct 2013, 1 from Oct 2014)

Contact us if you are interested.

Improve the world by applying session types to everything!
Honda, Yoshida and Carbone [2008] developed a theory of multi-party session types, generalising from the original two-party (binary) theory.

Multi-party session types provide a methodology for the design of communication-based systems, and there is an increasing amount of tool support.
Example from HYC 2008

informal design

---

1. \( B_1 \rightarrow S : \ s\langle \text{string}\rangle. \)
2. \( S \rightarrow B_1 : \ b_1\langle \text{int}\rangle. \)
3. \( S \rightarrow B_2 : \ b_2\langle \text{int}\rangle. \)
4. \( B_1 \rightarrow B_2 : \ b'_2\langle \text{int}\rangle. \)
5. \( B_2 \rightarrow S : \ s\{\text{ok} : B_2 \rightarrow S : \ s\langle \text{string}\rangle. S \rightarrow B_2 : \ b_2\langle \text{date}\rangle. \text{end}, \)
   \( \quad \text{quit} : \text{end}\} \)
Example from HYC 2008

1. B1 → S : s<string>.  
2. S → B1 : b₁<int>.  
5. B2 → S : s{ok : B2 → S : s<string>.S → B2 : b₂<date>.end,  
   quit : end}

Buyer 1:   s<string> ; b₁<int> ; b₂'!<int>

Buyer 2: b₂?<int> ; b₂’?<int> ; s⊕{ ok : s<string> ; b₂?<date> ; end,  quits : end }

Seller :  s?<string> ; b₁!<int> ; b₂!<int> ; s&{ ok : s?<string> ; b₂!<date> ; end,  
          quit : end }
Buyer 1: \( s!\langle\text{string}\rangle ; b1?\langle\text{int}\rangle ; b2'!\langle\text{int}\rangle \)

Buyer 2: \( b2?\langle\text{int}\rangle ; b2'?\langle\text{int}\rangle ; s \oplus \{ \text{ok} : s!\langle\text{string}\rangle ; b2?\langle\text{date}\rangle ; \text{end}, \text{quit} : \text{end} \} \)

Seller: \( s?\langle\text{string}\rangle ; b1!\langle\text{int}\rangle ; b2!\langle\text{int}\rangle ; s \& \{ \text{ok} : s?\langle\text{string}\rangle ; b2!\langle\text{date}\rangle ; \text{end}, \text{quit} : \text{end} \} \)

Local types are used for typechecking in an endpoint language

Global types are expressed in Scribble [Honda et al. 2007 – 2013] and the JBoss Savara project is developing associated tools.