A Session Type Provider

Compile-Time API Generation of Distributed Protocols with Refinements in F#

Rumyana Neykova    Raymond Hu    Nobuko Yoshida    Fahd Abdeljallal

Imperial College
London
Mobility Research Group

NEWS

SELECTED PUBLICATIONS

2018


Post-docs:
Simon CASTELLAN
David CASTRO
Francisco FERREIRA
Raymond HU
Rumyana NEYKOVA
Nicholas NG
Alceste SCALAS

PhD Students:
Assel ALTAYEVA
Juliana FRANCO
Eva GRAVERSEN
POPL 2008 Most Influential Paper Award

Kohei Honda, Nobuko Yoshida and Marco Carbone

Multiparty asynchronous session types
**Ocean Observatories Initiative**

**OOI aims:** to deploy an infrastructure (global network) to expand the scientists’ ability to remotely study the ocean

**Usage:** Integrate real-time data acquisition, processing and data storage for ocean research,…
Scribble – Proving a distributed design

1. All design work takes place in ABACUS, DCC’s enterprise architecture tool. This can export standard XMI files (an open standard for UMLs)

2. XMI is converted into OpenTracing format for consumption by managed service

3. OpenTracing files are combined to build a model in Scribble

4. Model holds types rather than instances to understand behaviour

5. Scribble compiler identifies inconsistency, change & design flaws

6. Issues highlighted graphically in Eclipse

7. Generate exception report and send back to DCC

www.estafet.com

Estafet Managed Service
Interactions with Industries

Strange Loop
SEPTEMBER 15-17 2016 / PEABODY OPERA HOUSE / ST. LOUIS, MO

Adam Bowen @adamnbowen · Sep 15
I didn't even know that session types existed an hour ago, but thanks to Nobuko Yoshida's great talk at #pwlconf, I want to learn more.

Nobuko Yoshida
Imperial College, London

DoC researcher to speak at Golang UK conference
by Vicky Kapogianni
20 July 2016

@nicholascwng rocking on @GolangUKconf about static deadlock detection in #golang #gouk16

The Golang UK Conference
Interactions with Industries

**F#unctional Londoners Meetup**

Session Types with Fahd Abdeljallal

6 days ago · 6:30 PM

Synopsis: Session types are a formalism to codify the structure of a communication, using types to specify the communication protocol used. This formalism provides the... LEARN MORE

---

**Distributed Systems vs. Compositionality**

Dr. Roland Kuhn
@rolandkuhn — CTO of Actyx

Current State

- behaviors can be composed both sequentially and concurrently
- effects are not yet tracked
- Scribble generator for Scala not yet there
- theoretical work at Imperial College, London (Prof. Nobuko Yoshida & Alceste Scalas)
Behavioural Type-Based Static Verification Framework for GO

Julian Lange
Nicholas Ng
Bernardo Toninho
Nobuko Yoshida
Go concurrency verification research at DoC grabs headline

A paper by DoC researchers at POPL on Go concurrency verification was featured in a tech blog and generates a buzz outside of the research community.

A paper by researchers at the department was recently featured in the morning paper, a blog by venture capitalist Adrian Colye, which summarises an important, influential, topical or otherwise interesting paper in the field of computer science every weekday in an easily digestible way by non-researchers. On the 2 Feb 2017 issue of the morning paper, It was highlighted as "the true spirit of POPL (Principles of Programming Languages)".
A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

tags: Concurrency, Programming Languages

**A static verification framework for message passing in Go using behavioural types** Lange et al., *ICSE 18*

*With thanks to Alexis Richardson who first forwarded this paper to me."

We’re jumping ahead to ICSE 18 now, and a paper that has been accepted for publication there later this year. It fits with the theme we’ve been exploring this week though, so I thought I’d cover it now. We’ve seen verification techniques applied in the context of **Rust** and **JavaScript**, looked at the integration of **linear types in Haskell**, and today it is the turn of Go!
Selected Publications 2017/2018

- [CC’18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
Selected Publications 2017/2018


[CC’18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.


A Session Type Provider

Compile-Time API Generation of Distributed Protocols with Refinements in F#

Rumyana Neykova
Imperial College London
United Kingdom

Raymond Hu
Imperial College London
United Kingdom

Nobuko Yoshida
Imperial College London
United Kingdom

Fahd Abdeljalil
Imperial College London
United Kingdom

Abstract
We present a library for the specification and implementation of distributed protocols in native F# (and other .NET languages) based on multiparty session types (MPST). There are two main contributions. Our library is the first practical development of MPST to support what we refer to as interaction refinements: a collection of features related to the refinement of protocols, such as message-type refinements (value constraints) and message-value dependent control flow. A well-typed endpoint program using our library is ranted to perform only compliant session I/O actions on the refined protocol, up to premature termination.

1 Introduction
Type providers [20, 27] are a .NET feature for a form of compile-time meta programming, designed to bridge between programming in statically typed languages such as F# and C#, and working with so-called information spaces—structured data sources such as SQL databases or XML data.

A type provider works as a compiler plugin that performs on-demand generation of types; it takes a schema for an external information space, and generates types that allow the data to be manipulated via a strongly-typed interface, with benefits such as static error detection and IDE auto-completion. For example, an instantiation of the in-built type provider for WSDL Web services [6] may look like

shots fired @zeeshanlakhani · Mar 12
Replying to @graydon_pub @dsyme
Awesome!

Brendan Zabarauskas @brendanzaub · Mar 12
Replying to @graydon_pub
This stuff fills me with hope!

Ryan Riley @panesofglass · Mar 12
Replying to @graydon_pub
This is amazing! I guess I need to switch
A Session Type Provider

Compile-Time API Generation of Distributed Protocols with Refinements in F#

Rumyana Neykova  Raymond Hu  Nobuko Yoshida  Fahd Abdeljallal

Imperial College
London
Part One

Type Providers
Type Providers

**Problem**: Languages do not integrate information
- We need to bring information into the language

---

**Types from data**: Making structured data first-class citizens in F#

Tomas Petricek
University of Cambridge
tomas@tomaspl.net

Gustavo Guerra
Microsoft Corporation, London
gustavo@codebeside.org

Don Syme
Microsoft Research, Cambridge
dsymef@Microsoft.com

**Abstract**
Most modern applications interact with external services and access data in structured formats such as XML, JSON and CSV. Static type systems do not understand such formats, often making data access more cumbersome. Should we give up and leave the messy world of external data to dynamic typing and runtime checks? Of course, not!

We present F# Data, a library that integrates external structured data into F#. As most real-world data does not come with an explicit schema, we develop a shape inference
Before Type Providers

OH NO!

WHAT HAVE YOU DONE!?!?

With Type Providers

all data is typed
on-demand generation
autocompletion
background type-checking
WorldBank Type Providers

```csharp
let data = WorldBank.GetDataContext()
```

Source

IDE/PROGRAM

Compiler

Type Provider
Useful for structured data?

👍

How about structured communication?
A generalisation to distributed protocols requires
- a notion of **schema for structured interactions** between services
- an understanding of how to extract the **localised behaviour** for each service
Part Two
Session Types
Multiparty Asynchronous Session Types

Kohei Honda  
Queen Mary, University of London  
kohei@dcs.qmul.ac.uk

Nobuko Yoshida  
Imperial College London  
yoshida@doc.ic.ac.uk

Marco Carbone  
Queen Mary, University of London  
carbonem@dcs.qmul.ac.uk

Abstract
Communication is becoming one of the central elements in software development. As a potential typed foundation for structured communication-centred programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focusing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centred applications. Presented as a typed calculus for mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types retain a friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual services (Carbone et al. 2006, 2007; WS-CDL; Sparkes 2006; Honda et al. 2007a). A basic observation underlying session types is that a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or session. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

As an example, the following session type describes a simple business protocol between Buyer and Seller from Buyer's viewpoint: Buyer sends the title of a book (a string), Seller sends a quote (an integer). If Buyer is satisfied by the quote, then sends his address (a string) and Seller sends back the delivery date (a date); otherwise it quits the conversation.

\[
\texttt{!string; ?int; @
(\texttt{ok :string; ?date; end, quit : end})}
\] (1)
Session Types

- **Protocol Validation**
  
  (int) \texttt{from C to S};
  
  (bool) \texttt{from S to C};

- **Program Verification**
  
  \texttt{runB c = let (x, c’) = receive c in send true c’}

A system of \textit{well-behaved processes} is free from deadlocks, orphan messages and reception errors.
Useful for structured data?

Data Type providers bring information into the language as strongly tooled, strongly typed

How about structured communication?

Session Type providers bring communication into the language as strongly tooled, strongly typed
Our Solution: Session Type Providers

```plaintext
type Prot = STP<“Prot.scr”, C>
let s = new Prot().Init()
  s.

Div(x:int, y:int) from C to S;
Res(z:float) from S to C;
```
Our Solution: Session Type Providers

```
type Prot = STP<"Prot.scr", C>
let s = new Prot().Init()
```

Session Type Provider
Our Solution: Session Type Providers

```
type Prot = STP<"Prot.scr", C>
let s = new Prot().Init()
s.send(S, Div, 6, 3)
```

Div(x:int, y:int) from C to S;
Res(z:float) from S to C;
Our Solution: Session Type Providers

type Prot = STP<"Prot.scr", C>
let s = new Prot().Init()
s.send(S, Div, 6, 3)

Div(x:int, y:int) from C to S;
Res(z:float) from S to C;

State3 State1.receive(S Role, Res label, Buf<float> f)
Our Solution: Session Type Providers

```plaintext
type Prot = STP<"Prot.scr", C>
let s = new Prot().Init()
   s.send(S, Div, 6, 3)
   .receive(S, Res, y)

Div(x:int, y:int) from C to S;
Res(z:float) from S to C;
```

Session Type Provider
Our Solution: Session Type Providers

```plaintext
type Prot = STP<"Prot.scr", C>
let s = new Prot().Init()
  s.

Div(x:int, y:int) from S to C;
Res(z:float) from S to C;
```

Session Type Provider
Our Solution: Session Type Providers

type Prot = STP<“Prot.scr”, C>

let s = new Prot().Init()
    s.send(S, Div, 6, “hello”)

Div(x:int, y:int) from C to S;
Res(z:float) from S to C;

Wrong payload
Our Solution: Session Type Providers

```
type Prot = STP<"Prot.scr", A>
```

Wrong protocol

```
Div(x:int, y:int) from C to S;
Res(z:float) from S to C;
```
Session Type providers bring communication into the language as strongly tooled, strongly typed.
global protocol C: role S, role C{
    choice at C {
        Div(x:int, y:int) from C to S;
        Res(z:float) from C to S;
        do Calc(C, S);
    } or {
        Add(x:int, y:int) from C to S;
        Res(z:int) from S to C;
        do Calc(C, S);
    } or {
        Sqrt(x:float) from C to S;
        Res(y:float) from S to C;
        do Calc(C, S);
    } or {
        Bye() from C to S;
        Bye() from S to C;
    }
}
global protocol Calc(role S, role C) {
    choice at C {
        Div(x:int, y:int) from C to S; @y!=0
        Res(z:float) from S to C;
        do Calc(C, S);
    } or {
        Add(x:int, y:int) from C to S;
        Res(z:int) from S to C;
        do Calc(C, S);
    } or {
        Sqrt(x:float) from C to S; @x>0
        Res(y:float) from S to C;
        do Calc(C, S);
    } or {
        Bye() from C to S;
        Bye() from S to C;
    }
}
Scribble with refinements

```
global protocol Calc(role S, role C){
  choice at C {
    Div(x:int, y:int) from C to S; @y!=0
    Res(z:float) from S to C;
    do Calc(C, S);
  } or {
    Add(x:int, y:int) from C to S;
    Res(z:int) from S to C;
    do Calc(C, S);
  } or {
    Sqrt(x:float) from C to S; @x>0
    Res(y:float) from S to C;
    do Calc(C, S);
  } or {
    Bye() from C to S;
    Bye() from S to C;
  }
}
```

interaction refinement $E$

$E :: x | n | \text{true} | \text{false} | E + E | \Theta E | f(E_1, \ldots, E_n) \$

$\Theta :: \text{and} | \text{or} | = | < | > | + | * \quad \Theta :: \text{not} | -$
Part Three

A Session Type Provider
What do you get from a session type provider?

**Session Types**
- A statically well-typed endpoint program will never perform a non-compliant I/O action w.r.t. the source protocol.

**Safety**

**Type Providers**
- compile-time generation
- background type checking & auto-completion
- a platform for tool integration (e.g. protocol validation)

**Usability**

**Interaction refinements**
- runtime enforcement of constraint
- implicitly send values that can be inferred (safe by construction)
- do not send values that can be locally inferred

**Reliability**

38
A Session Type Provider (Architecture)

The type provider framework is used for tool integration.
<table>
<thead>
<tr>
<th>Model</th>
<th>Properties</th>
<th>CFSM</th>
<th>F# Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>
Bounded model checking as a validation methodology [FASE’17]

Safety Properties:
- reception-error freedom
- orphan-message freedom
- deadlock freedom
<table>
<thead>
<tr>
<th>Model</th>
<th>Properties</th>
<th>CFSM</th>
<th>F# Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Refinement satisfiability**

**Refinement progress**
Refinement satisfiability

- check if the conjunction of all formulas is satisfiable
e.g. \((\text{and } (> y (+ x 1))(< y 4)(> x 3))\)

```
1(x:int) from A to B; @x>3
choice at B {2() from B to A;}
or {3(y:int) from B to A; @y>x+1 and y<4}
```

Checks if all execution paths are reachable

```
1(x:int) from A to B; @x>3
choice at B {2() from B to A;}
or {3(y:int) from B to A; @y>x+1 and y>4}
```
Refinement satisfiability

- check if the conjunction of all formulas is satisfiable
  e.g. \((\text{and} (\text{> } y (\text{+ } x \text{ 1}))(\text{< } y \text{ 4})(\text{> } x \text{ 3}))\)

```
1(x:int) from A to B; @x>3
choice at B {2() from B to A;}
  or {3(y:int) from B to A; @y>x+1 and y<4}
```

```
1(x:int) from A to B; @x>3
choice at B {2() from B to A;}
  or {3(y:int) from B to A; @y>x+1 and y>4}
```
Refinement progress

- check if formula is satisfiable for all preceding solutions
  e.g. \((\forall ((x \text{ Int}) (y \text{ Int}))(\rightarrow (> x 3)(\text{or} (< x y)(> x y))))\)

```
1(x:int) from A to B; @x>3
2(y:int) from A to B;
choice at B {3() from B to A; @x>y}
```

**Ensures that at any output point in the protocol implementations there will be always some values for which the formula holds**
## Refinement progress

- check if formula is satisfiable for all preceding solutions
  
e.g. \((\forall (x \text{ Int})(y \text{ Int}))(\Rightarrow (> x 3)(\lor (< x y)(> x y))))\)

<table>
<thead>
<tr>
<th>Model</th>
<th>Properties</th>
<th>CFSM</th>
<th>F# Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ((x:\text{int})\text{ from } A \text{ to } B; \ @x&gt;3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 ((y:\text{int})\text{ from } A \text{ to } B;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>choice at B {3()\text{ from } B \text{ to } A; @x&gt;y}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or {4()\text{ from } B \text{ to } A; @x&lt;y}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 1 \((x:\text{int})\text{ from } A \text{ to } B; \ @x>3\) | | | | |
| 2 \((y:\text{int})\text{ from } A \text{ to } B;\) | | | | |
| choice at B \{3()\text{ from } B \text{ to } A; @x>=y\} | | | | |
| or \{4()\text{ from } B \text{ to } A; @x<y\} | | | | |

| 1 \((x:\text{int})\text{ from } A \text{ to } B; \ @x>3\) | | | | |
| 2 \((y:\text{int})\text{ from } A \text{ to } B;\) | | | | |
| choice at B \{3()\text{ from } B \text{ to } A; @x>y\} | | | | |
| or \{4()\text{ from } B \text{ to } A; @x<y\} | | | | |

1(x:int) from A to B; @x>3
2(y:int) from A to B;
choice at B {3() from B to A; @x>y}
or {4() from B to A; @x<y}
(x:T1) from A to B; (y:T2) from B to C; (z:T3) from C to A;
global protocol Calc(role S, role C){
choice at C {
    Div(x:int, y:int) from C to S; @y!=0
    Res(z:float) from S to C;
    do Calc(C, S);
} or {
    Bye() from C to S;
    Bye() from S to C;
}
}
Map each state to a class

Map each transition to a method, e.g:
- send method
- receive method
type State2 =
    member send: C*Res*float → State1

type State3 =
    member send: C*Bye → State4

type State4 =
    member finish: unit → End
global protocol Calc(role S, role C) {
    choice at C {
        Div(x:int, y:int) from C to S; @y!=0
        Res(z:float) from S to C; @z=x/y
        do Addeer(C, S);
    } or {
        Bye() from C to S;
        Bye() from S to C;
    }
}
type State1 =
member branch: unit → ChoiceS1

type Div = interface ChoiceS1
  member receive: int*int → State2

type Bye = interface ChoiceS1
  member receive: → State3

type State2 =
  member send: C*Res*float → State1

type State3 =
  member send: C* Bye → State4

type State4 =
  member finish: unit → End
type State1 =
    member branch: unit → ChoiceS1

type Div = interface ChoiceS1
    member receive: int * int → State2

type Bye = interface ChoiceS1
    member receive: → State3

type State2 =
    member send: C*Res*float → State1

type State3 =
    member send: C*Bye → State4

type State4 =
    member finish: unit → End
let rec calcServer (c:Calc.State1) =

    match c.branch() with
    | :? Calc.Bye as bye ->

    | :? Calc.Div as div ->

    calcServer c1
```fsharp
let rec calcServer (c:Calc.State1) =
    let x, y = new Buf<int>(), new Buf<int>()
    match c.branch() with
    | Calc.Bye as bye ->
        bye.receive(C)
        .send(C, Bye)
        .finish()
    | Calc.Div as div ->
        let c1 = div.receive(C, x, y)
        .send(C, Res, x.Val/y.Val)
        calcServer c1
```

send

- serialise payload
- constraints as lambda functions

- quotations
- splicing

manage and use TCP sockets
type Prot = STP<"Prot.scr", C>
let s = new Prot().Init()
s.send(S, Div, 6, 3)
A statically well-typed STP-endpoint program will never perform a non-compliant I/O action w.r.t. the source protocol.
Compile-time performance

Example (role)          | #LoC | #States | #Types | Gen (ms)
------------------------|------|--------|--------|--------
2-Buyer ($B_2$) [13]    | 16   | 7      | 7      | 280    
3-Buyer ($B_3$) [5]     | 16   | 7      | 7      | 310    
Fibonacci (S) [14]      | 17   | 5      | 7      | 300    
Travel Agency (A) [24]  | 26   | 6      | 10     | 278    
SMTP (C) [14]           | 165  | 18     | 29     | 902    
HTTP (S) [3]            | 140  | 6      | 21     | 750    
SAP-Negotiation (C) [18]| 40   | 5      | 9      | 347    
Supplier Info (Q) [24]  | 86   | 5      | 25     | 1582   
SH (P)                  | 30   | 12     | 15     | 440    

Type and Code Generation (no refinements)
Protocol checking (no refinements)
Type and Code Generation (with refinements)
Protocol checking (with refinements)

API Generation does not impact the development time
Run-time performance

- Runtime overhead due to:
  - branching, runtime checks, serialisation
  - The performance overhead of the library stays in 5%-7% range
  - The performance overhead of run-time checks is up to 10%-12%
Future work and Resources

Framework Summary
- Type-driven development of distributed protocols
- Support for refinements on message interactions
- …ask me for more supported features

Future Work
- Static verification of refinements
- Partial model checking
- Support for erased type providers (event-driven branching)

Resources:
- Session type provider: https://session-type-provider.github.io
- Scribble: http://scribble.doc.ic.ac.uk/
- MRG: mrg.doc.ic.ac.uk
Thank you!

MAY THE F#ORCE BE WITH YOU

ALWAYS
Q & A

Questions

Answers

parse -> analyse -> pretty print
Questions

- documentation on the fly
- non-blocking receive
- explicit connections

Answers

- recompilation on protocol change
- online vs offline mode
- support by any .Net language

Check the tool for more features:

parse-> analyse -> pretty print
Related work

Related works on Interaction Refinements

- A theory of design-by-contract for distributed multiparty interactions [CONCUR’12]
- Linearly refined session types [LINEARITY’12]
- A concurrent programming language with refined session types. [BEAT’13]
- Certifying data in multiparty session types [JLAMP’17]

- no implementation
- based on syntactic checks
- developed for pi-calculus