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@tomas-p.net

Gustavo Guerra  
Microsoft Corporation, London  
gustavo@codebeside.org

Don Syme  
Microsoft Research, Cambridge  
dsyme@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

```fsharp
let doc = Http.Request("http://api.openweathermap.org/?q=NYC")
match JsonValue.Parse(doc) with
| Record(root) ->
  match Map.find "main" root with
  | Record(main) ->
    match Map.find "temp" main with
    | Number(num) -> printfn "Lovely %f!" num
    | _ -> failwith "Incorrect format"
  | _ -> failwith "Incorrect format"
| _ -> failwith "Incorrect format"
```
Before Type Providers

OH NO!
WHAT HAVE YOU DONE!?!

With Type Providers

```haskell
let doc = Http.Request("http://api.own.org/?q=NYC")
match JsonValue.Parse(doc) with
| Record(root) →
  match Map.find "main" root with
  | Record(main) →
    match Map.find "temp" main with
    | Number(num) → printfn "Lovely %f!" num
    | _ → failwith "Incorrect format"
    | _ → failwith "Incorrect format"
    | _ → failwith "Incorrect format"
```

```haskell
type W = JsonProvider("http://api.own.org/?q=NYC")
printfn "Lovely %f!" (W.GetSample().Main.Temp)
```

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

let data = WorldBank.GetDataContext()

data.
- Countries
- Regions
- ServiceLocation
  - _GetCountries
  - _GetCountry
  - _GetRegion
  - _GetRegions

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

How about structured communication?
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

\[ !\text{string}; ?\text{int}; ?(\text{ok} : !\text{string}; ?\text{date}; \text{end}, \text{quit} : \text{end}) \]  

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.

\[ !\text{string}; ?\text{int}; ?(\text{ok} : !\text{string}; ?\text{date}; \text{end}, \text{quit} : \text{end}) \]
A system of 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
let s = new Prot().Init()

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

```plaintext
type Prot = STP<"Prot.scr", C>
```

Session Type Provider
Our Solution: Session Type Providers

```plaintext
let s = new Prot().Init()

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

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

send

State2 State1.send($ Role, Div label, int x, int y)
Constraints: y! = 0
```
Our Solution: Session Type Providers

```plaintext
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

te 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;
Our Solution: Session Type Providers

```
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;
```
Our Solution: Session Type Providers

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

Wrong payload

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

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

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

Wrong protocol

Session Type Provider
Session Type providers bring communication into the language as strongly tooled, strongly typed.
global protocol Calc(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;
        Bye() from S;
    }
}
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;
} or {
    Bye()
}
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.

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

**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

**Safety**

**Usability**

**Reliability**
A Session Type Provider (Architecture)

The type provider framework is used for tool integration
Bounded model checking as a validation methodology [FASE’16]

Safety Properties:
- reception-error freedom
- orphan-message freedom
- deadlock freedom
Refinement satisfiability

Refinement progress
Refinement satisfiability

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

```fsharp
definition
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

```fsharp
definition
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. \((\text{and} \ (\gt \ y \ (+ \ x \ 1))(\lt \ y \ 4)(\gt \ x \ 3))\)

1(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. \((\forall (x:\text{Int})(y:\text{Int})) (\Rightarrow (> x 3) \lor (< x y)(> x y)))\)

Makes sure 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. \((\forall (x \in \text{Int})(y \in \text{Int})) ((\Rightarrow (x > 3) \lor (x < y) \Rightarrow (x > y)))\)

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

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

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

✅

✅/❌
(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
Model

Properties

CFSM

F# Type

Code

1

C?Bye

C?Div(int,int)

C!Res(float)

2

3

C!Bye

4

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
let rec calcServer (c:Calc.State1) =

    match c.branch() with
    | Calc.Bye as bye ->
    | Calc.Div as div  ->

    calcServer c1
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 cl = div.receive(C, x, y)
        .send(C, Res, x.Val/y.Val)
        calcServer cl
send

- serialise payload
- manage and use TCP sockets
- constraints as lambda functions

- quotations
- splicing
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

<table>
<thead>
<tr>
<th>Example (role)</th>
<th>#LoC</th>
<th>#States</th>
<th>#Types</th>
<th>Gen (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Buyer (B₁) [13]</td>
<td>16</td>
<td>7</td>
<td>7</td>
<td>280</td>
</tr>
<tr>
<td>3-Buyer (B₁) [5]</td>
<td>16</td>
<td>7</td>
<td>7</td>
<td>310</td>
</tr>
<tr>
<td>Fibonacci (S) [14]</td>
<td>17</td>
<td>5</td>
<td>7</td>
<td>300</td>
</tr>
<tr>
<td>Travel Agency (A) [24]</td>
<td>26</td>
<td>6</td>
<td>10</td>
<td>278</td>
</tr>
<tr>
<td>SMTP (C) [14]</td>
<td>165</td>
<td>18</td>
<td>29</td>
<td>902</td>
</tr>
<tr>
<td>HTTP (S) [3]</td>
<td>140</td>
<td>6</td>
<td>21</td>
<td>750</td>
</tr>
<tr>
<td>SAP-Negotiation (C) [18]</td>
<td>40</td>
<td>5</td>
<td>9</td>
<td>347</td>
</tr>
<tr>
<td>Supplier Info (Q) [24]</td>
<td>86</td>
<td>5</td>
<td>25</td>
<td>1582</td>
</tr>
<tr>
<td>SH (P)</td>
<td>30</td>
<td>12</td>
<td>15</td>
<td>440</td>
</tr>
</tbody>
</table>

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
Questions

parse -> analyse -> pretty print

Answers
Questions

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

Check the tool for more features:

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

parse-> analyse -> pretty print