Fluid Types

Statically Verified Distributed Protocols with Refinements

Fangyi Zhou    Francisco Ferreira
Rumyana Neykova    Nobuko Yoshida
Quick Primer on Session Types
Concurrenty

Shared Memory

Message Passing
A

(1) -> “Hello”

B

(2) <- 42

Send string
Receive int
Done

Duality

Send string
Receive int
Done
1. A -> C "Hello"
2. C -> B 42
3. A -> B "Hello"
4. B -> A 42
1. A -> C "Hello"
2. C -> B 42
3. A -> B "Hello"
4. B -> A 42

Send C string
Send B string
Receive B int
Done

Receive C string
Receive B string
Send B int
Done
(1) A -> C “Hello”
(2) C -> B 42
(3) A -> B “Hello”
(4) B -> A 42

string from A to C;
int from C to B;
string from A to B;
int from B to A;

Global Protocol
Motivation
Example: a simple protocol

- Two kids are playing a game on the playground
- \textbf{A} tells \textbf{B} a number
- \textbf{B} tries to find a larger number

```java
protocol Playground (role A, role B) {
    initialGuess (int) from A to B;
    finalGuess (int) from B to A;
}
```

No guarantee whether this will be larger
Example: a simple protocol

- Two kids are playing a game on the playground
- A tells B a number
- B tries to find a larger number

```plaintext
protocol Playground (role A, role B) {
    initialGuess (x:int) from A to B @ x > 7;
    finalGuess (y:int) from B to A @ y > x;
}
```

Named Parameters

Assertions
Previously...

- Session Type Provider [Neykova et al. 2018]
  - Compile Time Type Generation in F#
  - Protocol validated during compilation
  - Refinements checked dynamically during execution

Workflow (Previously)

protocol Playground
  (role A, role B) {
    initialGuess (x:int)
      from A to B @ x > 7;
    finalGuess (y:int)
      from B to A @ y > x;
  }

let p =
  new Protocol().Init()
  in
  p.send(B, initialGuess, 42)
  .receive(B, finalGuess, y)
  .finish()
Workflow (Now)

Protocol with Refinements → Compile Time Refined Type Generation → Communication via Generated Refined API → Refinement Type Check
Overview

• Add refinements to generated types

• Check refinements with a type system extension
  • Extract F# code into a refinement calculus
  • Check satisfiability using external solver
What are refinement types?

- Build upon an existing type system
- Allow base types to be refined via predicates
- Specify data dependencies
- Example: Liquid Haskell [Vazou et al. 2014]

Refinement Calculus: $\lambda^H$

- STLC with refinement types
- Terms can be encoded in SMT-LIB terms
- Establishes a subtyping relation via SMT solver
Types in $\lambda^H$

• A base type

$$\{\nu : b \mid M\}$$

Base type $b$, value $\nu$ refined by term $M$

• A function type (dependent function)

$$(x : \tau_1) \rightarrow \tau_2$$

Variable $x$ can occur in the type $\tau_2$

c.f. Dependent Types $\Pi_{x : \tau_1} \tau_2(x)$

integers, booleans, …
Example

- The integer literal \(1\)
  - A possible type: \(\{\nu : \text{int} \mid \nu = 1\}\)
  - Another possible type: \(\{\nu : \text{int} \mid \nu \geq 1\}\)
  - Or more… \(\{\nu : \text{int} \mid \text{true}\}\)
- Solution: Bidirectional Typing
Bidirectional Typing

- Provides a more algorithmic approach
- Mutually inductive judgments
- Type Synthesis

\[ \Gamma; \Delta \vdash M \Rightarrow \tau \quad \text{Given } \Gamma, \Delta, M, \text{ find the type } \tau \]

*Not all terms are synthesisable*

- Type Check

\[ \Gamma; \Delta \vdash M \Leftarrow \tau \quad \text{Given } \Gamma, \Delta, M, \tau, \text{ determine if type is correct} \]
"Change of Direction" Rule

**Subtyping Judgment**

\[
\Gamma; \Delta \vdash \tau <: \tau' \quad \Gamma; \Delta \vdash M \Rightarrow \tau \quad \Gamma; \Delta \vdash \tau'
\]

**Well-formedness Judgment**

\[
\Gamma; \Delta \vdash M \Leftarrow \tau'
\]
Subtyping with SMT

- Encode refinements term into SMT-LIB
- Use SMT solver to decide validity

\[
\text{Valid}(\Gamma \land \Delta \land [M_1] \implies [M_2])
\]

\[
\Gamma, \Delta \vdash \{v : b \mid M_1\} <: \{v : b \mid M_2\}
\]
Encoding in SMT-LIB

Χ (A term Variable)  Χ (An SMT Variable)
Encoding in SMT-LIB

\((+ \ 1 \ 2)\)
Encoding in SMT-LIB

\[ x : \{ \nu : \text{int} \mid \nu + 2 = 5 \} \quad \Rightarrow \quad x + 2 = 5 \]
Encoding in SMT-LIB

Valid(\([\Gamma] \land [\Delta] \land [M_1] \implies [M_2]\))

Unsat(\([\Gamma] \land [\Delta] \land [M_1] \land \neg [M_2]\))
Subtyping with SMT

- Consider integer literal 1

- Synthesised type: \( \{ \nu : \text{int} \mid \nu = 1 \} \)

- Check subtype: \( \{ \nu : \text{int} \mid \nu = 1 \} <: \{ \nu : \text{int} \mid \nu \geq 1 \} \) ?

- Encode into logic: \( \text{SAT}((\nu = 1) \land \lnot(\nu \geq 1)) \) ?

- Use SMT solver: UNSAT
Subtyping with SMT

- Consider term $x + 1$ with context $x : \{ \nu : \text{int} | \nu \geq 1 \}$

- Synthesised type: $\{ \nu : \text{int} | \nu = x + 1 \}$

- Check subtype: $\{ \nu : \text{int} | \nu = x + 1 \} \prec \{ \nu : \text{int} | \nu \geq 2 \}$

- Encode into logic: $\text{SAT}((x \geq 1) \land (\nu = x + 1) \land \neg(\nu \geq 2))$?

- Use SMT solver: $\text{UNSAT}$
Generating Types

• Scribble validates protocol and generates CFSM
• Type Provider converts CFSM into F# code
• New: Adding refinements in types
From Protocol to CFSM
(Scribble)

protocol Playground (role A, role B) {
    initialGuess (x:int) from A to B @ x > 7;
    finalGuess (y:int) from B to A @ y > x;
}

Projection to role A

protocol Playground (role A, role B) {
    initialGuess (x:int) from A to B @ x > 7;
    finalGuess (y:int) from B to A @ y > x;
}
From Protocol to CFSM (Scribble)

protocol Playground (role A, role B) {
  initialGuess (x:int) from A to B @ x > 7;
  finalGuess (y:int) from B to A @ y > x;
}

Projection to role A

\[ s_0 \to B \text{ ! initialGuess}(x: \text{ int}); x > 7 \to s_1 \]
\[ s_1 \to B \text{ ? finalGuess}(y: \text{ int}); y > x \to s_2 \]
From CFSM to $\lambda^H$ (Type Provider)

\[
\begin{align*}
\emptyset & \quad x : \{ \nu : \text{int} \mid \nu > 7 \} \\
\text{type State0} = \{} & \quad \text{type State1} = \{} \\
& \quad x : \{ \nu : \text{int} \mid \nu > 7 \}; \\
& \quad \} \\
& \quad y : \{ \nu : \text{int} \mid \nu > x \}; \\
& \quad \}\end{align*}
\]

\[
\begin{align*}
\text{type State2} = \{} & \quad \text{type State2} = \{} \\
& \quad x : \{ \nu : \text{int} \mid \nu > 7 \}; \\
& \quad y : \{ \nu : \text{int} \mid \nu > x \}; \\
& \quad \} \\
\end{align*}
\]
From CFSM to $\lambda^H$

(Type Provider)

![Diagram with states and transitions]

- Initial state $s_0$: $B ! \text{initialGuess}(x: \text{int}); x > 7$
- State $s_1$: $B ? \text{finalGuess}(y: \text{int}); y > x$
- State $s_2$

Type definitions:

- **State0**: `{}`
- **State1**: `{ x: {v: int | v > 7} }`
- **State2**: `{ x: {v: int | v > 7}, y: {v: int | v > x} }`

**Initial Guess Function**: `initialGuess : (st: State0) -> (x: {v: int | v > 7}) -> State1`
From CFSM to $\lambda^H$

(Type Provider)

$\lambda^H$

$\forall \beta \geq 7$

Final Guess:

$\text{finalGuess : (st: State1) -> (State2 * \{v:int|v>\text{st.x}\})}$

Type State 0:

$\text{type State0 = {}}$

Type State 1:

$\text{type State1 = {}}$

$\text{x: \{v:int|v>7\};}$

$\text{}}$

Type State 2:

$\text{type State2 = {}}$

$\text{x: \{v:int|v>7\};}$

$\text{y: \{v:int|v>x\};}$

$\text{}}$

Transition:

$s_0 \xrightarrow{\text{B ! initialGuess(x: int); x > 7}} s_1 \xrightarrow{\text{B ? finalGuess(y: int); y > x}} s_2$
From CFSM to $\lambda^H$
(Type Provider)

Initial Guess:

Initial state: $s_0$

- B ! initialGuess($x$: int); $x > 7$

Final state: $s_2$

- B ? finalGuess($y$: int); $y > x$

Type State0 = {}  

- type State1 = {
  x: {v:int|v>7};
}

Type State2 = {}  

- type State2 = {
  x: {v:int|v>7};
  y: {v:int|v>x};
}

Initial Guess:

- initialGuess : (st: State0) -> (x: {v:int|v>7}) -> State1

Final Guess:

- finalGuess : (st: State1) -> (State2 * {v:int|v>st.x})
One Last Step…

- Typecheck the program with refined types
  - Extract F# expressions to terms in $\lambda^H$
  - Use F# Compiler Services to obtain AST
  - Check whether API usage is correct w.r.t. refinements
Future Work

• Support recursion in protocols

• Complete meta-theory for refinements in MPST
  • End to end meta-theory

• Support more features in refinement calculus
Thank you!