Pabble: Parameterised Scribble for Parallel Programming

Nicholas Ng    Nobuko Yoshida
Imperial College London

9 Jan 2014
Outline

Introduction

Session C: Static type checking with Scribble

Pabble: Parameterised Scribble

Conclusion and future work
Motivation

- Parallel architectures
  - Efficient use of hardware resources
  - eg. Multicore processors, computer clusters
  - Difficult to program (correctly)
- Most common MPI error [Intel survey, SE-HPCS’05]
  - Communication mismatch (send-receive)
  - Communication deadlocks
Session Types for parallel programming

Session C  Static type checking against Scribble protocol
Pabble  Source code generation from parametric protocol

- Express communication topologies as sessions/protocol
- Guarantees
  - Communication safety
  - Deadlock freedom
Approach 1 - Session C programming

- Top-down approach
- Multiparty session types (MPST)
  - Communication: duality
  - Communication safety, deadlock freedom by typing
Session C programming: Key reasoning

1. Design protocol* in global view
2. Automatic projection to endpoint protocol, algorithm preserves safety
3. Write program according to endpoint protocol
4. Check program conforms to protocol
5.⇒ Safe program by design
Session C runtime

- Message passing API
  - Fast P2P communication
  - Lightweight
- Designed to be simple
  - Resembles Scribble
  - Some collective ops support
Session C runtime: Examples

Iteration and message passing

1 rec X {
2    int to A;
3    continue X;
4 }

1 rec Y {
2    int from B;
3    continue Y;
4 }

API (simple conditional)

1 while (i<3) {
2    int val = 42;
3    send_int(&val, 1, A);
4 }

1 while (i<3) {
2    int val;
3    recv_int(&val, 1, B);
4 }
Session C runtime: Examples

**Iteration and message passing**

```c
1  rec X { 
2      int to A; 
3      continue X; 
4  }
```

```c
1  rec Y { 
2      int from B; 
3      continue Y; 
4  }
```

**API (chained conditional)**

```c
1  while (outwhile(A, i<3)) { 
2      int val = 42; 
3      send_int(&val, 1, A); 
4  }
```

```c
1  while (inwhile(B)) { 
2      int val; 
3      recv_int(&val, 1, B); 
4  }
```
Session C runtime: Examples

Directed choice

Scribble

```
1 choice to B {
2   LABEL0(int) to B;
3 } or {
4   LABEL1(int) to B; }
```

API

```
1 if (i<3) { // Choice from
2   outbranch(B, LABEL0);
3   send_int(B, 12);
4 } else {
5   outbranch(B, LABEL1);
6   send_char(B, 'A'); }
1 // Choice to
2 switch (inbranch(A, &label)) {
3   case LABEL0:
4     recv_int(A, &ival); break;
5   case LABEL1:
6     recv_char(A, &cval); break; }
```
Session Type checking

- Static analyser
- Does source code conform to specification?
- Extract session type from code
  - Based on usage of API
  - Based on program flow control
- Compare w/ endpoint protocol
Session Type checking: Asynchronous optimisation

- Protocols designed safe
- Naive impl. inefficient
- Asynchronous impl.
  - Non-blocking send
  - Blocking receive
- Overlap send/recv operations
- Safety by async. subtyping [Mostrous et al., ESOP’09]
Summary (1/2): Session C programming framework

- Approach: Safety by type checking
- Protocol-based parallel programming framework
- Developer friendly Session Types as protocols
- Implementation with custom API
- Guarantees communication safety, deadlock free by design
Approach 2: MPI Pabble Code generation approach

- Scaling: More practical parallel programming
- Message Passing Interface (MPI) is standard API
- Associate **Parameterised** MPST with MPI
  - Type representation (protocol)
    - Pabble: Parameterised Scribble
    - Scribble roles with indices
  - Type check/extraction from source code
    - Parameterised (dependent) type checking non-trivial
    - MPI deductive verification
    - Related: next talk this session
- Our solution: Code generation from Pabble protocols
global protocol Ring(role Worker1, role Worker2, role Worker3, role Worker4) {
    rec LOOP {
        Data(int) from Worker1 to Worker2;
        Data(int) from Worker2 to Worker3;
        Data(int) from Worker3 to Worker4;
        Data(int) from Worker4 to Worker1;
        continue LOOP;
    }
}
Pabble: Parameterised Scribble

- **Parameterised Scribble** extension
- Role parameterisation by indices
- Grouping: Single endpoint protocol for parameterised roles
- Parametric extension of Scribble
  - `foreach`, recursion with loop index binding
  - `if`, conditional execution (multiple roles in single endpoint)
  - Role index calculation, design based on [Concurrency: state models and Java programs, Magee and Kramer, 2006]
- Scalable: Supports unbounded number of roles (for some cases)
Indexed interaction statement

Global protocol

1. Data(int) from Worker[i:1..9] to Worker[i+1];

Endpoint protocol

- All Workers share an endpoint protocol
- statements are executed conditionally (by index)

1. if Worker[i:2..10] Data(int) from Worker[i-1];
2. if Worker[i:1..9] Data(int) to Worker[i+1];
Example: Ring topology in Pabble

```
1 global protocol Ring(role Worker[1..N]) {
2   rec LOOP {
3     Data(int) from Worker[i:1..N-1] to Worker[i+1] ;
4     Data(int) from Worker[N] to Worker[1] ;
5     continue LOOP;
6   }
7 }
```
Ring protocol: Worker endpoint

```
local protocol Ring at Worker[1..N](role Worker[1..N]) {
  rec LOOP {
    if Worker[i:2..N] Data(int) from Worker[i-1];
    if Worker[i:1..N-1] Data(int) to Worker[i+1];
    if Worker[1] Data(int) from Worker[N];
    if Worker[N] Data(int) to Worker[1];
    continue LOOP;
  }
}
```
MPI code generation

- Sessions and MPI: Similar program structure
  - Pabble also single-source multiple-endpoints
  - Parameterised role index = MPI ranks
- Pabble vs. core MPI primitives, e.g.
  - P2P: Send, Receive
  - Collective ops: Scatter, Gather, All to All
Ring protocol: Simplified MPI code

1. `MPI_Init(&argc, &argv);`
2. `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
3. `MPI_Comm_size(MPI_COMM_WORLD, &N);`
4. `#pragma pabble loop`
5. `while (1) { // rec LOOP`
6. `// if Worker[i:2..N] Data(int) from Worker[i-1];`
7. `if (2<=rank && rank<=N} MPI_Recv(.., MPI_INT, rank-1, Data, .. );`
8. `// if Worker[i:1..N-1] Data(int) to Worker[i+1];`
9. `if (1<=rank && rank<=N-1} MPI_Send(.., MPI_INT, rank+1, Data, .. );`
10. `// if Worker[1] Data(int) from Worker[N];`
11. `if (rank==1} MPI_Recv(.., MPI_INT, N, Data, .. );`
12. `// if Worker[N] Data(int) to Worker[1];`
13. `if (rank==N} MPI_Recv(.., MPI_INT, 1, Data, .. );`
14. `}`
15. `MPI_Finalize();`
Summary (2/2): MPI code generation from Pabble

- Approach: Safety by code generation
- Generate MPI backbone
  - Communication-correct
- Pabble indexed roles to rank
- Supports MPI collective ops
Conclusion: Session-based safe parallel programming

- Define global protocol in Scribble
  protocol P
  {int from A to B}

- Project into endpoint protocol
  protocol P at A
  {int to B}

Code generation approach

- Generate MPI code
  ```
  int main() {
    MPI_Send(buf, cnt, RANK_B, MPI_INT, ...);
  }
  ```

- Implement program
  ```
  int main() {
    MPI_Send(buf, cnt, RANK_B, MPI_INT, ...);
  }
  ```

Type checking approach

- Implement program
  ```
  int main() {
    calc(buf, cnt);
    send_int(B, 42);
  }
  ```

Static type checking

- Check implementation conforms with endpoint protocol at compile time

- Communication safety
- Deadlock free
Ongoing and future work

- Extract/verify Session Types from MPI
  - Can we infer global types from the endpoint MPI programs?
  - Extract Pabble from MPI using simulation
    - Masters project (2013)
  - Deductive verification of MPI using VCC
    - Collaboration with FCUL [EuroMPI’12, PLACES/BEAT2’13]
- Applying methodology in different environments
  - Software-Hardware communication (eg. FPGA, Maxeler)
  - Parallel code generation & parallelisation via AOP
  - Reconfigurable hardware (FPGA) code generation & transformation