Session-Typed Communication

Frank Pfenning

Computer Science Department
The Unreasonable Effectiveness of Types

- Types capture fundamental abstractions
  - $A \rightarrow B$, a function from $A$ to $B$
  - $A \times B$, a pair with components of type $A$ and $B$
  - $1$, the unit type with one element
  - $A + B$, a tagged sum with a component of type $A$ or $B$
- Can be combined to describe rich forms of data and computation
  - $\text{List } A = (\text{Nil} : 1) + (\text{Cons} : A \times \text{List } A)$
  - $\text{map} : (A \rightarrow B) \rightarrow (\text{List } A \rightarrow \text{List } B)$
- Types also provide new abstractions
  - Guarantee representation independence (in some languages)
Types as Interfaces

- Between programmer and code: expressing and verifying intent
- Between compilation source and target: relying on program invariants
- Between library and client: module interface
What About Processes?

• Types would seem to be ideal
• We need to generalize
  • From sequential, memory-related abstractions
  • To concurrent, message-passing abstractions
  • Giving up neither theoretical guarantees nor pragmatic usability
• What are these concurrent, message-passing abstractions?
  • Session types [Honda’93]
  • Logical underpinnings of session types [Caires & Pf’10][Caires, Pf, Toninho’16]
  • Multiple implementations (Scribble, Mungo, SILL, CC0, ...)

CyLab
Carnegie Mellon University
Security and Privacy Institute
Example: A Store (Stack or Queue)

- Interaction protocol
  - Client: insert; \( x; \) \textit{recurse}...
  - Client: delete
    - Provider: none; close
    - Provider: some; \( x; \) \textit{recurse}...

- This protocol should be expressed by a type!
- Start with an \textbf{external choice} between ‘insert’ and ‘delete’
- After ‘delete’, the provider has an \textbf{internal choice} between ‘none’ and ‘some’
Session Types, Abstractly

<table>
<thead>
<tr>
<th>Type</th>
<th>Provider action</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$ ::=</td>
<td>$&amp;{\ell : A_\ell}_{\ell \in L}$</td>
<td>receive some $k \in L$</td>
</tr>
<tr>
<td></td>
<td>$\oplus{\ell : A_\ell}_{\ell \in L}$</td>
<td>send some $k \in L$</td>
</tr>
<tr>
<td></td>
<td>$A \rightarrow B$</td>
<td>receive channel $c : A$</td>
</tr>
<tr>
<td></td>
<td>$A \otimes B$</td>
<td>send channel $c : A$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
<td>terminate</td>
</tr>
<tr>
<td></td>
<td>$\forall x : \tau. A$</td>
<td>receive $v : \tau$</td>
</tr>
<tr>
<td></td>
<td>$\exists x : \tau. A$</td>
<td>send $v : \tau$</td>
</tr>
</tbody>
</table>

$\text{stack}_A = \&\{ \text{ins} : A \rightarrow \text{stack}_A, \text{del} : \oplus\{ \text{none} : 1, \text{some} : A \otimes \text{stack}_A \} \}$
What We Learned at CMU

• Four implementations
  • SILL (functional + concurrent) [Toninho’15] [Griffith’16]
  • CC0 (imperative + concurrent + sharing) [Willsey et al.’16] [Balzer & Pf.’17]
  • SS (pure pipelined concurrency with time & work analysis) [Das et al.’18]
  • Nomos (functional + concurrent, with work analysis for digital contracts; in progress) [Das et al.’19]

• Static type checking is feasible, with good feedback to programmer

• Dynamic type and contract checking is feasible [Jia et al.’16] [Gommerstadt et al.’18]
Examples and Case Studies

- Parallel & concurrent data structures & algorithms
  - Queues, stacks, treaps, mergesort, fork/join, futures, ...
- Pipelined algorithms
  - Odd/even sort, prime sieve, balanced ternary numbers, ...
- Distributed algorithms
  - Dining philosophers, cigarette smokers
  - Dynamic deadlock detection, record-and-replay
  - Raft distributed consensus
Research Questions

- Leverage types for security properties (in addition to safety)
  - Information flow, secure multi-execution
  - Timing channels
- Leverage typing discipline for correctness proofs
  - Distributed systems code
  - Concurrent algorithms
  - Shared memory implementations
- How do we connect different levels of abstraction?