Bismuth: A Programming Language for Distributed, Concurrent & Mobile Systems

Alex Friedman (CS/PW)
Advisors: Professor Rose Bohrer (CS), Professor Yunus Telliel (PW)

Research Problem
From cloud computing to machine learning and the rise of IoT devices, it is apparent that the future of computing will occur in a more distributed and concurrent manner than ever before [1, 2]. However, such programs are challenging to write as few languages are designed for these tasks. As such, this project developed Bismuth: an expressive new language designed to enable the communication of distributed, concurrent, and mobile tasks while retaining correctness guarantees and being accessible to a general audience of programmers. To accomplish this, I developed and used a new framework for the rapid audience-centered prototyping of programming languages.

Design Framework
1. Initial Design: Establish Audience & Purpose
   Statement of Motivations
   Case Study of Languages
   Create Initial Theory

2. Revise Design: Evaluate Expressiveness & Design Impact
   Corpus Study
   Can express audience needs?
   Language impact on tasks?

3. Assess Design: Analysis of Results & Conclusion
   Either: reject, revise, or deem ready for intensive methods

Conclusions & Future Work
• Bismuth has the potential to be expressive for a wide variety of distributed tasks—representing 5/7 audience tasks with at most minor simplifications; the remaining require modifications that could be reasonably addressed by future work (parallel and closeable channels).
• Bismuth's consistent approach to distribution made programs that would otherwise require libraries easy to write.
• Bismuth's correctness properties make some tasks (such as shared state) as they require additional code for the language to verify them as correct. This could be addressed through expanding what the language can understand as correct.
• Bismuth's syntax conceals pragmatic information about what processes do due to the complexity of protocols. Future work could explore ways of making the syntax more communicative.

Sample Program
Database :: c : Channel!IntChoice!
get : <Key: Option<Value>>, ...
set : <Key: Value>,
lock : <Key: IntChoice>

Map<Key, Value> data;
accept(c) {
  offer c {
    get => c.send(data[c.recv()])
    set => c.send(data[c.recv()])
    lock => 
      Key k = c.recv();
      Option<Value> opt = data[k];
      match opt {
        Empty => { c.send($Value$); }
        Value v => { c.send(data[k]); }
      }
  }
}
...
var db = exec Database;
Channel!{get : <Key: Option<Value>>, ...
setRq ...} rqs = ...
accept(rqs) {
  acceptWhile(setRq, true) (more(db); link(setRq, db))
  more(db)
  link(rqs, db)
  accept(writes) { more(db); link(setRq, db) }
  weaken(db)

Bismuth

Syntax
• Types
  P, Q ::= Program
  Channel ::= Parent/Child Cycle
  Channel ::= Parallel Process
  Program ::= Channel

• Syntax
  Parent/Child Cycle
  Parallel Process
  Option
  Linear and Non-linear Resources

Contributions & Features
• By using Classical Linear Logic, there is no need to distinguish between parent/child channel & both linear and non-linear resources are supported
• Asynchronous: Write operations are non-blocking
• Intermixed used of multiple channels within each process
• Linear resources are easily accessible within loops while maintaining correctness guarantees
• Through Accept While Loops, protocols of duration controlled by a remote process do not have to be recognized all at once
• Multiple processes can communicate over the same channel via protocol manipulations
• Acyclic graph distribution structure allows for flexibility and deadlock freedom

Proof Overview
Progress: If \((G ; E ; P) \text{ ok}\) then either \((G ; E ; P) \text{ done}\) or there exists \((G' ; E ; P')\) such that \((G ; E ; P) \Rightarrow (G' ; E ; P')\).
Preservation: If \((G ; E ; P) \text{ ok}\) and \((G ; E ; P) \Rightarrow (G' ; E ; P')\), then \((G' ; E ; P') \text{ ok}\).
Type Safety, Deadlock Freedom: Inherent given above proofs.

References

MQP Paper  PW Poster  Web Compiler  Website