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 to 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

Bismuth

Sample Program

```
Database :: c : Channel<!ExtChoice<get: +Key;-Option<Value>,
                                    set: +Key;+Value,
                                    lock:+Key;IntChoice<
                                                -Value;+Value,
                                                +Value>>> {

  Map<Key, Value> data;
  accept(c) {
    offer c
      | get => c.send(data[c.recv()])
      | set => data[c.recv()] = c.recv()
      | lock => {
          Key k = c.recv();
          Option<Value> opt = data[k];
          match opt
            | Empty => { c[+Value]; }
            | Value v => { c[-Value;+Value];
                            c.send(data[k]); }
          data[k] = c.recv();
        }
  }
}
...
var db = exec Database;
Channel<!(get : +Key;-Option<Value>, ...)> rqs = ...
Channel<!(+Key;+Value)> setRq = ...

accept(rqs) {
  acceptWhile(setRq, true) {more(db); link(setRq, db)}
  more(db)
  link(rqs, db)
}
accept(writes) { more(db); link(setRq, db) }
weaken(db)
```

Syntax

τ ::=	*Types*	P, Q ::=	*Program*
Channel<ρ>	Channel Type	s_b	Base Statement
Program<ρ>	Program Type	$P; Q$	Sequence
τ_b	Base Type	c.send(e)	Send
ρ ::=	*Protocols*	while(e){P}	While Loop
$+\tau$	Receive	c.case(P, Q)	External Choice
$-\tau$	Send	$c[i]$	Internal Choice
$?\rho$	Why Not	more(c)	Unfold
$!\rho$	Of Course	weaken(c)	Weaken
$\rho; \rho$	Sequence	accept(c){P}	Accept Loop
ExternalChoice<ρ, ρ>	External Choice	acceptWhile(c, e){P}	Accept While Loop
InternalChoice<ρ, ρ>	Internal Choice	exit	Exit Program
		e ::=	*Expressions*
		x	Variable
		c.recv()	Receive
		exec P	Execute Program
		e_b	Base Expression

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**

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.**

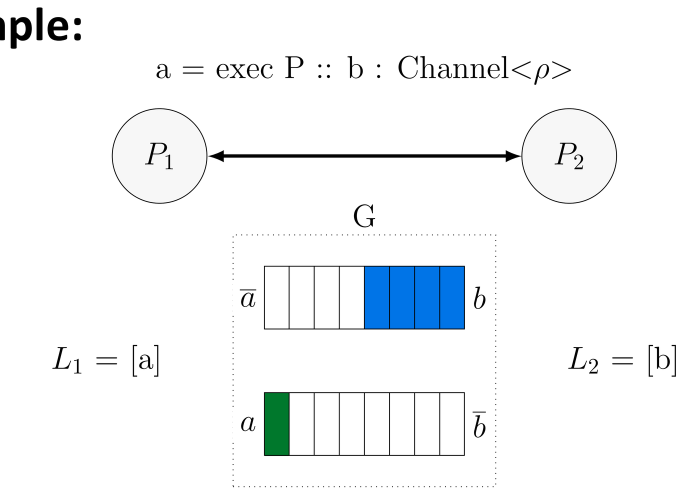
Proof Overview

Progress: If $(G; \vec{L}; \vec{P})$ ok then either $(G; \vec{L}; \vec{P})$ $done$ or there exists $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ such that $(G; \vec{L}; \vec{P}) \mapsto (G'; \vec{L}'; \vec{P}')$.

Preservation: If $(G; \vec{L}; \vec{P})$ ok and $(G; \vec{L}; \vec{P}) \mapsto (G'; \vec{L}'; \vec{P}')$, then $(G'; \vec{L}'; \vec{P}')$ ok.

Type Safety, Deadlock Freedom: Inherent given above proofs.

Configuration Example:



References

[1] Lindley, S., Morris, J.G. (2015). A Semantics for Propositions as Sessions. In: Vitek, J. (eds) Programming Languages and Systems. ESOP 2015. Lecture Notes in Computer Science, vol 9032. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-46669-8_23

[2] Akshitha Sriraman. "Enabling Hyperscale Web Services". PhD thesis. University of Michigan, 2021. doi: https://dx.doi.org/10.7302/2847.

MQP Paper

PW Poster

Web Compiler

Website