A Rhetorical Framework for Programming Language Design

Alex Friedman (CS/PW)

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

Introduction & Background

From cloud computing to machine learning and the rise of IoT devices, computing requires the coordination of distributed and concurrent programs more than ever before; however, such programs are challenging to write as traditional languages are not designed to express these kinds of tasks.

To help address this, I created Bismuth: a new programming language for distributed and concurrent tasks designed to be accessible to a general audience of programmers. As **existing language design frameworks** are either **high-cost & driven by user feedback** to proposed designs or l**ow-cost & driven by the designer's opinions,** to accomplish this, **I developed and used a new low-cost and audience-centered framework** for the rapid prototyping of programming languages **based on viewing computer languages as a rhetorical medium that programmers communicate within.**

Rhetorical Code Studies

Audience: Who uses the language

- Languages vary dramatically from general purpose (C++, Java, Python, etc.) to Excel, animation software, and more.

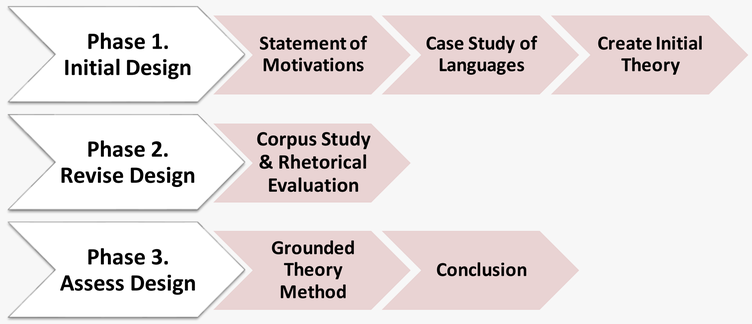
Metaphors: How we conceptualize the world & approach problems

- In coding, can be seen as the meaning of syntactic elements and the abstractions they allow users to create.
- Programs are easier to write when the task is easily mapped to the language's metaphorical system.

Procedural Rhetoric: Claims made by the rules of a system/procedure

- Languages are defined by rules that describe when each operation is valid.
- Unintentional effects of rules make such systems challenging to author.

Framework & Methods



Case Study: Bismuth

Background

With most languages designed around the traditional view of sequential computation and existing theories for distributed languages being mathematically terse, in developing Bismuth, I needed to determine what concepts would be helpful to users and how to represent them in an accessible manner—making its development a good test of this framework.

Findings

- Bismuth has the potential to express many audience tasks—representing 5/7 of the corpus tasks with at most minor simplifications, and the remaining limitations could be reasonably addressed by future work.
- Through using classical logic, Bismuth removes the need to distinguish each end of a channel which allows its protocol syntax to more closely resemble established computer science metaphors—making it easier to work with.
- Correctness properties allows for automatic handling of tedious tasks and the elimination of errors/bugs—allowing programmers to focus on communicating the novel computations they wish to express.
- Bismuth's limited number of rules makes expressing certain programs challenging (such as shared state)—even when, as a user of the language, we may be able to correctly reason about a program's validity.
- Bismuth's protocol syntax conceals what processes do by communicating data types without a means to name what the data represents.

Intuitionistic vs Classical Logic

```
Channel<((A⊗(B⊸C⊸D)⊸⊥)⊗1>
```

```
Channel<-A;+B;+C;-D>
```

Bismuth Prototype vs Traditional Notation

```
max :: c : Channel<!(+num);Option<num>)> {
  Option<num> optNum = Empty
  accept(c, 1) { optInt = c.recv() }

  match optNum
    |  Empty => {
       ✖ accept(c) {num n = c.recv() }
         c.send(optNum)
       }
    | num n => {
         accept(c) { n = Max(n, c.recv()) }
         c.send(n)
       }
}
```

```
Option<num> max(num[] numbers) {
  if numbers.length == 0 { return Empty }


  num n = numbers.pop()
  for(num i : numbers) { n = Max(n, i) }
  return n
}
```

Notes:

1. Code meaning roughly equivalent based on lines unless shown otherwise by arrow.
2. X represents dead code that is required to ensure program type checks.
3. Bismuth code finds the maximum number in a stream; Traditional does so in an array.

Sample Improvements

```
ExtChoice<Error, A;ExtChoice<Error; B;...>>
```

```
Closeable<A;B;...>
```

```
Channel<+Channel<A>; +Channel<B>>
```

```
Channel< a : A | b : B>
```

```
Channel<ExtChoice<A, B>> c = ...
c.case(
  <case for c : Channel<A>>,
  <case for c : Channel<B>>
)
```

```
Channel<ExtChoice<a : A, b : B, a2 : A>> c;
offer c
  | a => ...
  | a2 => ...
  | b => ...
```

Conclusions & Future Work

- This framework allowed me to critically examine Bismuth and learn about its ability to express common tasks in its domain.
- While results are less granular and generalizable than other frameworks, they are fast and easy to attain—making rapid iterations possible.
- Future work will be needed to verify the success of this framework and Bismuth; however, both seem promising in their applicability and ability to make their respective domains more accessible

References

[1]
[2]
[3]
[4]
[5]

MQP Report

Website

CS Poster