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 [1]; 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 and user-centered' or 'low-cost and designer-centered', to accomplish this, I developed a **low-cost yet audience-centered framework** for the rapid prototyping of programming languages. This works by viewing **computer languages as a rhetorical medium**—thus enabling us to evaluate the communicative and expressive potential of various language designs.

Rhetorical Code Studies

Despite the common perception, programming languages have **inherent rhetorical properties** including:

Audience: Who uses the language for what purpose

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

Metaphors: How we imagine and conceptualize the world

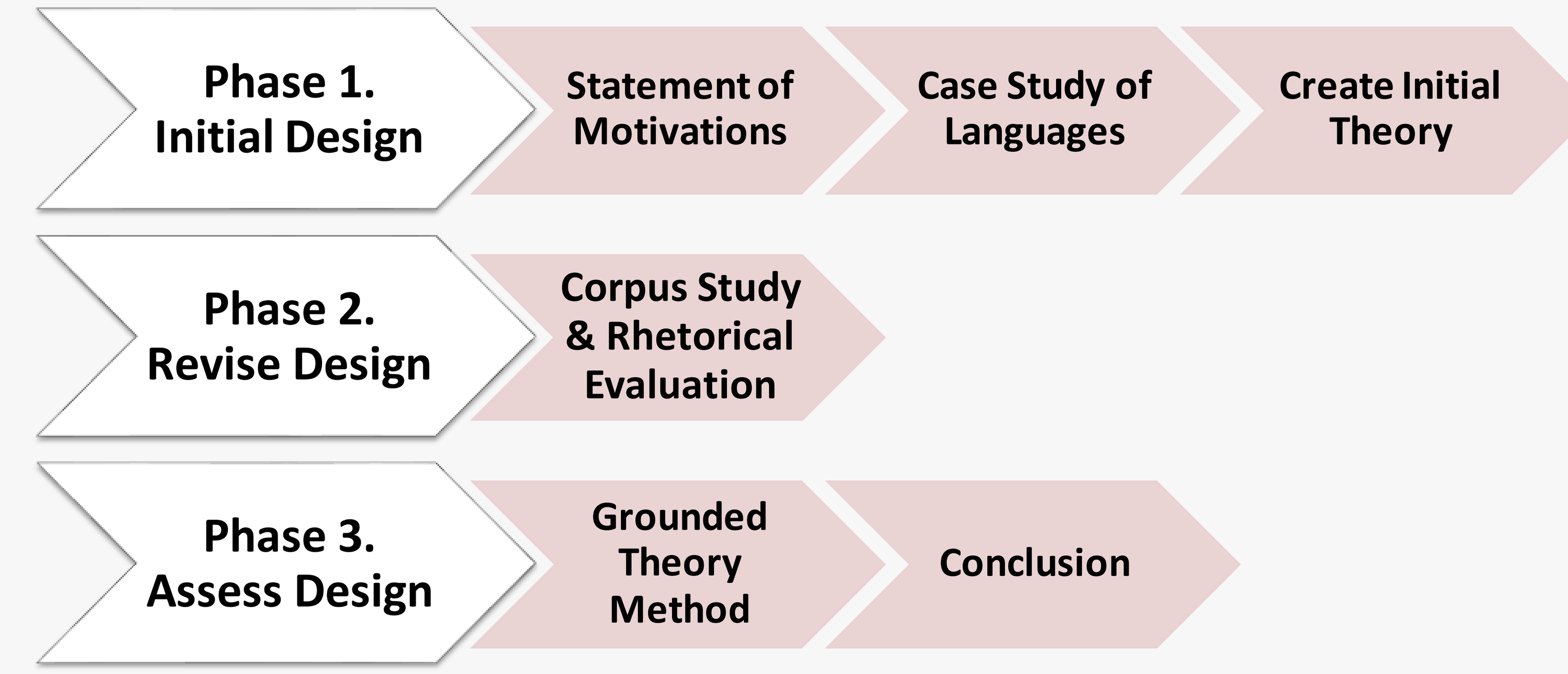
- The meaning of syntactic elements & the abstractions they allow users to create.
- Programming is easier when tasks can be easily conceptualized with the language's metaphors [2].

Procedural Rhetoric: Claims made by the rules of a programming language

- Meaning is produced by procedures rather than individual human actions.
- Unintentional effects of rules make such systems challenging to author.

Proposed Framework & Methods

In order to connect programming languages to the study of rhetoric, I developed the following language design framework:



Case Study: Bismuth

Background

Most languages have been designed with the traditional view of sequential computation and existing theories for distributed languages are often 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 my 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.
- Bismuth's protocol syntax conceals what processes do by communicating data types without a means to name what the data represents.
- 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.

Intuitionistic [3] vs Classical Protocols

```
(A⊗(B⊸C⊸D)⊸⊥)⊗1
```

```
-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
}
```

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] Lindley, S., Morris, J.G. "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] Green, T.R.G., and Petre, M. "Usability analysis of visual programming environments: a 'cognitive dimensions' framework." Journal of Visual Languages & Computing 7.2 (1996): 131-174.

[3] Mazurak, K., and Zdancewic, S. "Lolliproc: to concurrency from classical linear logic via Curry-Howard and control." ACM Sigplan Notices 45.9 (2010): 39-50.

MQP Report

CS Poster

Compiler

Website