Communicating with Process Calculus

Creating a Low-cost Audience-Centered Language Design Framework
&
Distributed, Concurrent and Mobile Programming Language for a General Audience

A Major Qualifying Project
submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
degrees of Bachelor of Science in Computer Science
and Bachelor of Science in Professional Writing.

By:
Alex Friedman

Advisors:
Professor Rose Bohrer, Computer Science
Professor Yunus Telliel, Professional Writing

August 24, 2022 – May 3, 2023

Abstract

This project introduces Bismuth: a general purpose and higher-order programming language for distributed concurrent and mobile systems. Through a combining classical linear logic, asynchronous communication, and a novel approach to resource management, Bismuth enables the expression of useful concepts other theories cannot while maintaining type safety. To accomplish this while ensuring that Bismuth would be accessible and a communicative tool for a general audience of programmers, we created (as also described in this paper) and used a low-cost and audience-centered framework for designing programming languages—filling a gap in existing methods between being high-cost and user centered, or low-cost and designer-centered. Further information, including an extensive implementation of Bismuth can be found at the project's website: `https://bismuth-lang.org/`.

[git] • Branch: main @ 81beae1 • Release: (2023-04-11)

Acknowledgements

Thank you to Professor Bohrer and Professor Telliel for all of your help, advice, and guidance throughout this project. This project would have been impossible without your support, and I appreciate all of the time and energy that you contributed towards these ends.

I would also like to thank the faculty and staff of WPI's Computer Science Department for the providing opportunity and space for me to work on such projects.

Contents

List of Figures

List of Tables

List of Programs

1 Introduction

Every day, it is becoming increasingly important to perform complex and interconnected computations at a scale that has not been seen historically; from cloud computing to machine learning and the rise of IoT devices, it is apparent that the future of computing will occur in a much more distributed and concurrent manner than ever before [38, 25]. The growing importance of this phenomena, sometimes referred to as hyperscale computing, is further evident from the worldwide number of data centers dedicated to such tasks more than doubling in five years from 338 in 2016 to 700 in 2021 [39]. However, despite these trends in computing, few programming languages are designed for expressing these kinds of tasks. Consequently, writing such programs can be challenging as breakdowns often form between what the code says it will do and what happens when it gets executed. Not only does this force programmers to forego many compile-time correctness guarantees, because programs now depend on the exact timing of concurrent events, they can be incredibly challenging to test and debug. This can lead to the dangerous illusion of a program working correctly during testing but failing in production.

While, in recent years, newer languages (notably Rust and Julia) have gained popularity for helping to address these problems, their approaches are limited to in-memory concurrency or treating distribution as an asynchronous function call (as opposed to an interaction of systems). Similarly, while many theories (collectively known as process calculi) describe how distributed languages could work, these are often too mathematically terse for a general audience of programmers or limited in their expressiveness to satisfy theoretical properties. The goal of this project was to bridge this gap by creating a more expressive process calculus that a general audience of programmers could use to communicate distributed, concurrent, and mobile tasks while retaining correctness guarantees. As designing a programming language is is a complicated audience-dependent task for which existing design frameworks are either high-cost and user-centered or low-cost and designer-centered, to accomplish this, I developed a new low-cost and 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 potential audience impact of various designs. This information then helps inform if the next step of the research should be further revisions, using more intensive methods, or trying an entirely different approach.

Using this approach, I created Bismuth: an expressive new language designed to enable to the asynchronous communication of distributed, concurrent, and mobile tasks while being accessible to a general audience of programmers and retaining correctness guarantees (including type-safety and deadlock freedom). Based on its evaluation, Bismuth has the potential to be an incredibly expressive language with its main limitations being with representing unreliable channels and complex resource sharing. While these limitations are generally considered open questions in the field of process calculi, we see several paths forward to address these and are exploring them in ongoing research—thus making Bismuth a promising path towards making distributed computing easier and more accessible. In addition, an implementation of Bismuth (which has been extended to include arrays, algebraic data types, pointers, pass-by-reference functions, and the automatic deep copying of any arbitrary non-linear value) and online compiler can be found at the project's website: `https://bismuth-lang.org/`.

2 Background - Programming Language Theory

This section first provides an brief introduction to the predominant theories used by programming languages and the motivations behind these theories in Section 2.1. It then discusses how the programming languages influenced by these theories have gone about adding support for parallel and distributed code execution (Sections 2.2 and 2.3 respectively) and the challenges associated with these approaches. Next, the section provides an overview to competing theories of computation that are designed to model parallel and distributed computations in Section 2.4 before, finally, describing a few type systems that we can augment a process calculus with that help us reason about how programs in these languages function (Section 2.5).

2.1 Models of Computation

At its core, one of computer science's foundational principles is the concept that we can build mathematical languages that are capable of performing computations. Of all the possible languages we could build, programming languages typically fall into the category of Turing-complete languages. Turing-complete languages are particularly remarkable because they are universal models of computation. This means that they are able to run any possible algorithm—allowing them to even simulate all other languages. Because of this, they are said to have the same computational power as the only difference between Turing-complete languages is how they express and perform computation. This principle allowed Alonzo Church to develop the λ-calculus language in the 1930s which has since gone on to be the foundational theory for most modern programming languages. The main downside to this, however, is that if we ever wish to represent computation in a way that λ-calculus is unable to, then we must either build a new theory or extend λ-calculus to represent this form of computation. One such limitation of λ-calculus is that it primarily describes processes as something that, when evaluated, results in a value. This makes modeling distributed and concurrent systems challenging in λ-calculus as, for those programs, it can be useful to view processes and how their interactions over time work together to perform computations.

2.2 Traditional Challenges for Concurrency

Despite λ-calculus's short-term view of processes, over time, the growing need for our programming languages to support interactions between concurrent computations has lead many programmers to create project-specific workarounds based on *forking* (Section 2.2.1) and *shared-memory concurrency* (Section 2.2.2).

2.2.1 Forking

Forking is when a process duplicates itself then typically uses control flow to make the original and duplicate process each execute different code. This, however, poses several challenges. First, because the two processes are completely separate, to enable any kind of interaction or cooperation between the processes, we must implement an Inter-process communication (IPC) protocol. This can be challenging as many languages require developers to implement IPC protocols themselves. This requires the developer to determine how to parse and represent data in the protocol. While languages can typically aid the developer in accomplishing the latter task through a concept called *serialization*, languages typically have limits on what data can be serialized. In addition, because many of these protocols run on untyped data streams, the developer must do so without any

compiler correctness guarantees. Given these additional steps to get forking to work, if we wish to use it in our programs, then we either have to design our programs around it ahead of time or restructure existing code to add it.

2.2.2 Shared-Memory Concurrency

Shared-memory concurrency (typically achieved through *multithreading*) is when we allow multiple programs (or threads) to share the same memory despite running independently. The major benefit of this approach is that we do not need to use an IPC protocol as programs can directly communicate through their shared memory; however, this also leads to the major downsides of shared-memory concurrency[1]: because each program/thread shares the same memory, we can end up in circumstances where two processes perform operations that overwrite or otherwise conflict with each other—a problem known as a race condition. To resolve such problems, the programmer must identify all regions where this could possibly happen and ensure that there is sufficient *mutual exclusion* between them to prevent *race conditions* while taking care to ensure that the programs do not enter *deadlock*. These issues are particularly challenging to resolve as, by definition, these are undecidable problems—meaning that there is no way for a computer to automatically identify all possible cases where this could occur and report them to the programmer. Thus, the programmer is required to spend a great deal of time and effort scrutinizing the codebase in hopes of finding and remediating such problems. The challenge of this task is then further exacerbated by the fact that any changes to the code could require the programmer to completely redo this task.

2.3 Traditional Challenges for Distribution

As it stands, programming languages typically provide rudimentary mechanisms for a programmer to write protocols to connect programs running on different computers, but they lack the expressiveness to aid the programmer in this process. In many ways, building a distributed application shares the same challenges as one would experience in using forking as described in Section 2.2.1 as both would require the use of similar inter-process communication protocols. Unlike forking, however, because distributed execution occurs across various devices, we are now presented with additional challenges including:

- Delayed and potentially unreliable communication.
- Different system resources and capabilities.

While a few programming languages directly support distributed execution, those that do have several major limitations. For instance, the Julia programming language's support for distributed execution is non-compositional and essentially acts like a thread pool except on the scale of compute clusters [13]. Similarly, while the Nuxt 3 and Next.JS JavaScript frameworks allow for full-stack and typed development of web applications and their APIs, these are ultimately more of a domain-specific languages for web development as they lack the versatility of a process calculus [33, 2].

[1] While purely functional programming languages do not have this problem as there is no global state and memory is immutable, they still have limitations when it comes to distribution as discussed in 2.3. Similarly, while there are a few thread-safe imperative languages (notably Rust) which avoid such issues by tracking the ownership of variables, they too have similar limitations for distributed computing.

2.4 Process Calculi

In contrast to λ-calculus, the family of languages known as process calculi provide us with ways to describe how concurrent processes interact to perform computations—typically in a distributed environment. As such, programming languages based on these theories should be able to provide programmers with the expressiveness to construct such systems without the challenges associated with building such a system in a λ-calculus based language. This section first provides an overview of one of the most popular process calculi: π-calculus in Section 2.4.1. Next, we discuss a π-calculus derivative known as Ambient calculus in Section 2.4.2. Finally, we discuss another form process calculus: Join-calculus in Section 2.4.3.

2.4.1 π-Calculus

As λ-calculus forms much of the basis for sequential computation, π-calculus is the basis for many theories of concurrent computation with many other process calculi having equivalent translations in π-calculus [10, 28].

In its original definition, π-calculus, created by Robin Milner, has a fairly rudimentary syntax containing just six production rules: create a channel, send a value over a channel, receive a value over a channel, run processes in parallel, repeat a process indefinitely, and terminate [37]. Using these few constructs, we are able to model incredibly complex interactions between concurrent processes. In fact, as later shown by Milner, this definition suffices to encode any λ-calculus in π-calculus—thus meaning π-calculus is a universal model of computation [29]. Thus, however, does not mean that π-calculus is without its criticisms—many of which extensions of the language seek to address.

Perhaps most immediately apparent of these criticism is that π-calculus lacks the notion of data types and thus the entire calculus is untyped[2]. This level of generality has led to ambiguity over how π-calculus should be interpreted. For instance, some use a higher-order interpretation of the language wherein channels and processes can be sent across other channels and other models disallow this [37]. In addition, as π-calculus lacks data types—including primitives—to perform computations in π-calculus, it is necessary to extend the calculus to include such operations. This, however, is usually done by allowing processes to perform λ-calculus operations—which raises more questions for how our language should work. For instance, by incorporating λ-calculus into π-calculus, one could ask if functions should be allowed in π-calculus. In doing so, we would have to consider:

- If functions are allowed, then, unlike λ-calculus wherein everything is a function, in process calculus, not everything is a process.
- If functions are allowed but, in reality, are just processes, then we need to revise our definition of process/function to make them the same thing without losing expressiveness.
- If functions are not allowed then, unless we are using a higher-order calculus, the calculus would have lost expressiveness.

Similarly, π-calculus's representation of channels also raises questions over how communications in the language should work. This is because channels in π-calculus are defined globally and can

[2]While later work has made typed versions of π-calculus[37], many derivations forego this in favor of developing their own type system (such as [28]).

appear visible to multiple process at once. Because of this, not only is it unclear who is able to communicate to whom, but it is also unclear how we should resolve situations where multiple processes are communicating across the same channel at the same time.

In addition to these instances of ambiguous program meanings, π-calculus also has limits on how it can describe programs. For instance, π-calculus dichotomizes processes into two types: those which are ephemeral (running one time) and those which are persistent (running infinitely many times). While this can still be used to model things like the interactions between clients and servers, it prevents us from modeling a circumstance where a server may loop for a duration of time before eventually proceeding onto other tasks (or even terminating). [37].

As such, despite π-calculus being a powerful mathematical foundation for parallel systems, communicating with it can become challenging due to its ambiguous interpretation and complex representation of traditionally simple interactions.

2.4.2 Ambient Calculus

Ambient calculus expands upon the traditional π-calculus framework by adding the concept of ambients which are defined as "a bounded place where computation happens" [10]. Because ambient calculus also requires all computations occur within an ambient, the language is able to describe processes that move in both the physical and virtual world. This framework helps limit ambiguity in the calculus by allowing us to clearly describe what resources move from one process to another in each communication. Additionally, because the calculus restricts ambient operations—including mobility—by a set of permissions called capabilities, communication in ambient calculus can be more realistic than that of π-calculus where any process can communicate with any other process via global channels.

Despite the benefits of ambient calculus, it still has a few drawbacks of its own. For instance, ambient calculus only models running processes—including when the ambient that the process resides in is moving [10]. In addition, the languages' own definitions of mobility are left untyped and ambiguous as many ambient operations allow for the introduction of resources with conflicting names.

2.4.3 Join-Calculus

Under the umbrella of process calculi, join-calculus is one alternative to π-calculus which seeks to address many of the same issues as ambient calculus[10]. Unlike π-calculus which focuses on how processes in a system communicate, join-calculus focuses on definitions (which describe, in a sense, the locality of a resource), and it uses join-patterns to describe how to execute and synchronize functions concurrently. For example, the "and" join-pattern could be used to require all functions to complete execution. In contrast, the "or" join-patter would allow functions to race—thus taking the value of the first function to terminate. While this provides a fairly intuitive calculus that almost reads as pseudocode—especially when compared to the other calculi—the main downside is that the join-pattern used to manage synchronization tends to either require parallel operations to occur prior to a function invocation (which limits expressiveness of the language) or the use of callbacks (which are notoriously challenging to use for complex communications)[45, 3].

2.5 Type Systems

While models of computation such as λ-calculus or process calculus allow us to perform calculations, they cannot tell us about what the program will do unless we run it, and, even then, we still may have a limited ability to reason about the program as it is possible that it will never terminate. By applying a type system to the language, however, we can impose restrictions on the language which allow us to determine some things about a program without even running it—including if it will have certain types of errors. In this section, we discuss two important type systems for parallel and distributed applications: session types in Section 2.5.1 and linear types in Section 2.5.2.

2.5.1 Session Types

While process calculi allow us to mathematically model distributed and concurrent systems, many do so over untyped communications channels and, as such, we forego the compile-time guarantees and additional mathematical reasoning that we would get from a typed language. One potential solution to this problem is to use session types which, in addition to providing us with typed channels, provide us with the guarantee that our program will never *deadlock*. In addition, with the exception of shared session types, they can also guarantee that that a program will not have *race conditions*. While the use of session types in a variety of process calculi has been explored in many papers including [28], their strict mathematical definition makes these languages overly restrictive. Notably, to provide us with these guarantees, session types require that each process only acts as the client to one server and that the communication between processes follows a rigid pattern of sequential operations (when one process sends, the other process must receive) [9]. While multi-party session types attempt to remediate this problem, they require an additional layer of complex global properties in order to ensure proper mutual exclusion between processes [43].

Because of this, programs that use session types are greatly limited in their ability to express programs and frequently introduce additional complexity into otherwise simple programs such as directly linking communications between to children of a server and having a client reply back to the server prior to the client's final operation [28].

2.5.2 Linear Types

Another useful type system for parallel and distributed programs is that of linear types. Derived from linear logic, in a linearly typed program, all variables must be used exactly one time. This allows us to ensure that each resource is only accessible by one program at a time—thus addressing many of the challenges, such as preventing race conditions, that we would have to deal with otherwise. In addition, because linear types only describe the requirements for how we use resources, it is possible to build a programming language that uses session types to describe how processes communicate and linear types to describe the location and existence of resources [15]. In some ways, ambient calculus takes a similar approach to this by using capabilities to specify how ambients can interact while also requiring that each ambient exists in only one place at a time.

Linear types, however, have two main downsides. The first is the requirement that types can only be used once. While this helps us write parallel and distributed programs, it also makes it impossible to implement data structures that many programs rely on (such as linked lists, trees, and graphs). This is because these data structures can require multiple references to the same resource which is impossible in a linear system. The second (and more minor) limitation is that we have to

use each resource once. While this can help us prevent excess memory use and computations, it is not always a practical restriction to impose[3][44].

2.6 Summary

Programming languages are powerful tools that allow us to describe any algorithmic computation with the differences between each language being how each program is expressed and processed—with many of our current languages being built off of sequential models of computation. Because of this, it can be incredibly challenging to use our current languages to represent concurrent and distributed programs—which are becoming incredibly important. To address these challenges, we can focus our attention to the field of process calculus as each language that falls under this classification is designed to reason about such parallel and distributed systems; however, in doing so, we find that these languages present their own challenges and limitations—even when augmented with a type system to improve our reasoning about them.

[3]While affine types (notably used by Rust) address this second issue by requiring types to be used no more than once, affine types are not directly relevant to this project.

3 Background - Programming Language Design

This section starts with a brief introduction into the history of programming languages which lays the foundation for the differences between programming languages and natural languages (Section 3.1). Given the split between natural languages and programming languages, the next section details some of these differences and the challenges they pose to the designers of programming languages (Section 3.2). In response to these challenges, the following section describes several frameworks designed to evaluate programming languages—with a particular emphasis on how the language may be experienced or viewed by a user (Section 3.3). Finally, building on these evaluation criteria, the section describes user-oriented design methodologies for programming languages (Section 3.4).

3.1 History of Programming Languages

In the early days of computing, computers were large machines that had to be manually reconfigured and rewired anytime that its functionality needed to change. In addition, these computers only understood basic operations (such as arithmetic), and it was up to the circuit designers to find ways of conveying complex tasks to these machines [41]. The modern notion of a general-purpose computer being able to run a variety of programs only started with the introduction of the von Neumann architecture in 1945. With this, binary representation of commands and values replaced rewiring as our primary method of communicating with computers. Three years later, in 1948, researchers devised a way of of converting human-readable textual representations of these commands into binary—thus leading to the creation of the first assembly language [41].

While this was a major step in making programming easier, writing programs in assembly language was still challenging. It was not until 1957 with the introduction of FORTRAN that modern high-level programming languages started to emerge. In these languages, code is represented in a manner which resembles natural language. This code is then read by a compiler which converts the language into machine code for the computer to execute. This made programs more concise and faster to write—hence leading to the rapid and widespread adoption of such languages [41].

Despite these advances towards a natural language representation of computer code, several obstacles prevent us from communicating programs with natural language. The first is that of societal context: the way humans communicate constantly changes depending on world events as well as their own background. The second is that of ambiguity: the natural language people communicate with has many sentences for which multiple interpretations exist depending on how the sentence is interpreted. As compilers need to know exactly what a human intends in order for it to produce the desired program for the computer to execute, these barriers have caused programming languages to become their own form of communication designed to allow humans to unambiguously communicate arbitrary processes to computers by representing them in terms of exact rules and behaviors [32].

3.2 Challenges Designing Programming Languages

As a consequence of programming languages becoming their own form of communication, how we interact with these languages differs from how we interact with other written forms of communication. Because of this, the way that we design programming languages requires us to consider what the language is doing in ways that differ from how we would think about other acts of communication such as writing. This section describes three such considerations for the design of a

programming language: Audience (Section 3.2.1), Metaphors and Abstractions (Section 3.2.2), and Procedural Rhetoric (Section 3.2.3).

3.2.1 Audience

Despite most programming languages having equal computational power[4], as "[each representation of a language] highlights some kinds information at the expense of obscuring other kinds"[17, 35] (and thus communicate concepts in different ways), the design of a programming language depends on who will be using it and what kinds of tasks they will be using it for.

3.2.1.1 Primary Audience Fundamentally, programming languages need to enable simultaneous communication between two distinct primary audiences, the computer and programmer, who would not otherwise be able to communicate easily [41]; to humans, the primitive binary-encoded representation of individual instructions understood by computers is illegible without great effort, and the natural language used by humans requires an understanding of ambiguity and societal context for which the computer lacks. Despite these differences, however, as the development of compilers has shown, as long as we can find an algorithmic mechanism for translating an input into something that the computer can process and execute, then we can consider that given input to be a programming language. Working within this constraint, the creators of programming languages have a lot of freedom in designing a language representation based around the human portion of their primary audience.

For example, esoteric programming languages, with the goal of representing code in unique (and sometimes cryptic) ways, have explored the possibility of representing code in various manners including through the creation of abstract art (such as the Piet programming languages) and through writing code such that it sounds like a play written by Shakespeare.

In contrast, at the other end of the spectrum, we have general-purpose programming languages that primarily rely on a fairly similar text-based representation of code. However, despite the similarities in their syntax—especially in the case of C, C++, Java, and C#—general-purpose programming languages allow programmers to describe concepts differently enough to make them particularly well suited for specific audience-dependent tasks. For instance, C and C++ are both low-level languages providing the programmer with direct control over many aspects of the computer—including how the program uses memory. While this makes them particularly well suited for tasks such as operating system development or embedded systems, it also requires that those using the language have a better understanding of how a computer functions than languages such as Java, Python, or C# which automatically manage memory for the programmer. While this means that the languages have to place restrictions on what kinds of programs are written—making them more suitable for different tasks and giving them the ability to be more easily run on systems with different architectures.

Between the esoteric languages (which are designed for creativity more than practicality) and the general-purpose languages (which are designed for practicality, but tend to follow similar design choices), being designed for a specific task, domain specific languages (or languages designed for non-programmers) frequently utilize language designs, such as having user interfaces with visual components, that help their audience achieve their specific goals while being impractical for other tasks. For instance, many languages designed to teach children to code (such as Scratch, Alice,

[4]They are able to perform the same tasks and express the same concepts as any other language with the main difference between programming languages being how concepts are represented

or HANDS) use draggable blocks which each represent some step in a program. As the goal of these languages is to teach algorithmic thinking, they do not include many of the concepts that would be required of the languages designed for software development, and they also do not have to worry about how efficiently the code would run [34, 41]. In contrast, other domain specific languages (DSLs) such as Microsoft Excel—targeting a very different audience than the aforementioned languages—may require specialized knowledge to use as their goal is to streamline the the work of professionals in specific fields.

3.2.1.2 Secondary Audience Beyond the language's primary audience of the programmer and computer, programming languages also enable communication between developers working on the same codebase. Thus, just as it is possible for us to write things in natural languages that others have a hard time reading, the designers of a programming language must make sure that the language is both easy for humans to write, and easy for humans to read [17]. To accomplish this, languages typically allow for programmers to write comments in their code—sections of the program that the computer ignores and instead serves as a way for people do document what a program does [7, 41]. While the designer of a language cannot prevent people from writing hard-to-read code (or require that the developer comments their code), by considering the human reader in designing a language, they can consider how various language features (such as comments, metadata[5], and other "secondary notations" [17]) may help improve the language's ability to communicate [17, 12].

3.2.2 Metaphors and Abstractions

While, traditionally, metaphors have been seen as a superfluous property of language which the creative expression of a concept through the use of words typically used to describe another similar concept for which one wishes to draw a comparison with [22], as argued by George Lakoff and Mark Johnson, metaphors are actually a pervasive component of everyday life with our thoughts and actions being influenced by the metaphors used to frame the world around us [23]. Thus, as the basis for our conceptual system, metaphors influence how we understand and approach the various tasks and problems that we encounter—including how we would navigate the challenge of mapping potentially real world problems into something understandable by a computer [17]. In this respect, despite computer science lacking a definition for what a metaphor is, at the language level, we can view metaphors as the concepts and abstractions that a programming language employs and makes possible as these ultimately bound how tasks within the language can be conceptualized.

Determining what metaphors a programming language should utilize, however, can be a challenging problem where each decision can have a substantial and long lasting impact. This is because, unlike natural languages, unlike natural languages which are constantly evolving through every person using it slightly differently and changing societal contexts, changes in programming languages happen slowly with only a small subset of the people using the language being responsible for making changes to the language (and everyone else building abstractions within the language's bounds). In addition, this changes can rarely respond directly to the current needs of programmers as language designers need to be careful to maintain interoperability between newer versions of the language and legacy codebases. As a consequence, over time, the metaphorical system of a programming language becomes cemented into the language, and the high barriers to creating a

[5]This could include language features such as assertions or state annotations in Obsidian [12] which serve more to limit or describe the function of a program to humans than something that the computer actually executes.

new programming language keep developers invested in using the existing solutions despite their flaws.

As such, despite a programming language's disconnect from ongoing societal contexts, it is critical that from its outset, that the language is expressive enough for programmers to be able to find ways of communicating new concepts in an otherwise static medium. This has impacts both on the syntactic design of programming languages as well as the higher-level concepts of how code is structured in a language.

At the syntax level, programming languages must describe the various possible operations and control flow statements in manners that are both consistent with how we would conceptualize these operations and that are easy for the programmer to recognize as individual syntactic elements with these meanings [7, 35, 17]. This is a concept sometimes referred to as as text structure knowledge (TS knowledge), and it has been shown to be one of the first steps programmers use to understand what a program that they are unfamiliar with does as it allows them to break down large problems into individual steps, procedures, and abstractions which, when combined, achieve the desired result [35, 7]. For example, we will consider how a for loop compares in a *C-style language* (such as C++ or Java) which is designed around human metaphors, versus the same code in LLVM IR which is designed around a computer's instruction set (Figures 1 and 2 respectively).

In the C-style languages, the concept of repetition is embraced as human metaphor where we want something to occur a number of times. Specifically, in Figure 1, a programmer would be able to look at this block of code, see the word 'for' and immediately know that whatever code between the following brackets will be repeated based on the conditions between the parentheses. From here, they can tell that, to initialize the loop, a new variable i, which stores a integer, will be created. The loop will then run so long as $i < 10$, and, upon each iteration of the loop, i will be incremented by 1.

Figure 1: For Loop in C++

```
1 for(int i = 0; i < 10; i++) {
2     //Do something
3 }
```

In contrast, LLVM IR, being designed around a computer's instructions, does not contain the notion of a loop as that is not something that a computer understands. As such, to represent the same concept, it must allocate the space for an integer (line 2), initialize that integer to have the value 0 (line 3), load the stored value (line 4), compare the value to 10 (storing the result as a new variable) on line 5, and then conditionally jumping to either the code that will be repeated (line 8) or the next step of the program (line 16). In addition, at the end of each iteration of the loop, we must increment the variable we are using in our comparison and repeat the whole process of comparing and conditionally jumping back to the loop to determine if we have completed the repetition (lines 9-14).

Figure 2: For Loop in LLVM IR

```
entry:
  %i = alloca i32, align 4
  store i32 0, i32* %i, align 4
  %i1 = load i32, i32* %i, align 4
  %0 = icmp slt i32 %i1, 10
  br i1 %0, label %loop, label %rest

loop:                                             ; preds = %loop, %entry
  %i2 = load i32, i32* %i, align 4
  %1 = add nsw i32 %i2, 1
  store i32 %1, i32* %i, align 4
  %i3 = load i32, i32* %i, align 4
  %2 = icmp slt i32 %i3, 10
  br i1 %2, label %loop, label %rest

rest:                                             ; preds = %loop, %entry
  ...

```

As the C-style languages recognize that the metaphor of a loop is so commonly used by humans and is easily translatable to computers, by incorporating loops as part of the programming language, they are able to make the code much easier for humans to read and much simpler for them to write [41].

Beyond the basic syntactic constructs of a programming language, however, designers must also recognize that people live in a changing world with changing needs and metaphors. As such, programming languages often provide developers with mechanisms to create their own abstractions within a language (such as objects and functions) which allow them to group elements in a way that allows the language to treat them as one entity that can be reused [17]. Over time, these abstractions (such as design patterns and data structures) have become so pervasive and integral to problem solving that programmers, recognizing these common patterns, structures, and concepts, are able to develop an understanding of what the code accomplishes in goal/action-oriented manner without paying attention to the individual steps in a program. This is sometimes known as plan knowledge (PK knowledge), and, as a programmer becomes more familiar with a codebase and computer science concepts, it slowly replaces TS knowledge as their mental model for comprehending code [35]. For example, while various different languages may provide different representations of a concept known as a map, by knowing that maps are a data structure which stores data of a specified value by correlating values to unique keys of another specified type, once they know how to recognize such a structure in a language, they are able to understand what kinds of operations the code may be achieving despite not being familiar with the entire codebase or the language's specific implementation of a map.

3.2.3 Procedural Authorship and Rhetoric

In designing a language, not only does one have to determine how things are said (the language's syntax and abstractions), but they also have to determine what kinds of things are allowed to be said in the language. In this respect, by designing the rules by which a programming language functions and expresses concepts, designers are making claims about how the world functions and what their

audience's needs are. This is captured through Ian Bogost's concept of procedural rhetoric which studies how the rules of a system interact to make such claims by both enabling forms of expression that one may not have thought of previously and restricting other forms of expression [5]. Janet Murray further expanded on this concept through introducing the concept of procedural authorship: "...writing the rules by which the texts appear as well as writing the texts themselves. It means writing the rules for the interactor's involvement, that is, the conditions under which things will happen in response to the participant's actions" [32]. This concept is particularly salient when considering programming language design because, not only does it contain the aforementioned notion that the rules of a system make claims about the world, but that there are also cases which go beyond this where the rules of a system also also establish a rhetoric inherent to any derivative works—much as how the rules and concepts in a programming language go on to influence the programs written in the language.

For example, despite the benefits of having abstractions in a programming language (as discussed in Section 3.2.2), every abstraction imparts some meaning about what kinds of programs can be written and how programs are written. As such, having too many metaphors in a programming language can result in it oversimplifying the concepts that developers may wish to express, and they may also make the language daunting to learn—thus potentially excluding portions of the language's audience [17, 1]. At the same time, however, there may also be concepts whose presence enables forms of expression that may otherwise be overlooked despite the potential for it to benefit the audience. For instance, Bogost points out that "...the very concept of returning a defective product is only made possible by the creation of rules that frame that very notion" [5]. In the context of programming languages, such a concept could be that of type safety wherein, by requiring that all variables are used in proper ways according to their types, a programming language is able to limit the number of runtime errors that can occur by preventing the construction of such programs.

While some of these design decisions may intentional or even desired by the language's audience, the designer must be aware of these decisions and their respective trade-offs as otherwise the language may be unintentionally limited in what concepts it can describe and how it describes concepts [7]. This can be particularly challenging as many of these impacts arise from the ways the various rules in our language (or the lack thereof) interact to form claims about how the world works [5], and, as such, they may not be apparent to the designer when evaluating the programming language. This is because, as with all aspects of our lives, our decisions in designing a language are influenced by our own conceptual system for which we are typically unaware of until something challenges our traditional way of thinking [23, 5].

Further enforcing this bias is the challenge of taking on the role of a procedural author as it requires us to approach authorship in manners contrary to how we traditionally would—despite the two seeming deceptively similar. According to Murray, "The notion of a procedural medium...takes some getting used to. We need time to grow accustomed to combining participation with immersion, agency with story, and to perceiving the patterns in a...fictional world" and that it challenges our traditional notion of authorship where works are fixed by presenting us with dynamic medium [32]. In the context of computer science, Murray's observation can describe the difference between traditional software development projects fields such as programming language development. When building an application, computer science best practices are analogous to providing a fixed authored experience to the user: the bounds of what users are allowed to do is rigidly determined by the programmers with any variation from the predefined use-cases being seen as a bug or exploit which must be patched. To this extent, users are to not be trusted in their own judgement and everything they do must be double checked to ensure that their actions will not break the system. In contrast,

programming languages seek to provide just enough guardrails to help those using the language eliminate certain kinds of errors while still being open enough to enable the expression of limitless problems within the language's domain—both known and unknown at the time of the language's development.

To help overcome this, while we evaluate language's theoretical properties to determine how computationally expressive the language is, these properties do not capture how concepts are expressed. Thus computer science theory alone cannot speak to if a language is conducive for (or a practical medium for) human communication. Because this, language designers need some other mechanism to illuminate their audience's needs so that they can determine if their language design fulfills these requirements while making minimal tradeoffs within the language's domain. In this respect, despite language designers authoring the environment in which the users of a language interact, they must do so in a manner which puts their audience first so that way people using the language are able to author their own works with the language as opposed to simply acting within a fixed set of performances instantiated by the designer's procedures. There is, however, no set way to go about this with procedural authorship traditionally focusing on where there is a clearer distinction between the author and later interactors [32].

3.3 Evaluating Programming Languages

Given the challenges in designing a programming language, several frameworks have been developed for the purpose of evaluating languages. This section provides an overview of three common approaches for this task: Cognitive Dimensions (Section 3.3.1), Quality Attributes (Section 3.3.2), and a Rhetorical Framework (Section 3.3.3).

3.3.1 Cognitive Dimensions

Green's Cognitive Dimensions Framework serves as a set of 13 "discussion-tools" (summarized in Figure 4) based on how users interact with programming languages. Thus, while they do not describe granular truths of language design, applying them to programming languages may help to highlight general strengths and weaknesses of the languages that the framework is applied to [17].

Prior to evaluating a language with this frameworks, one must establish an order of importance for each of the criteria in order to ensure an overall net benefit. This is because various design decisions may require making a trade-off between the criteria. At the same time, designers must also be wary of over optimizing the language to meet these criteria. This is because, despite allowing one to evaluate if a proposed change to a programming language would be beneficial, it cannot guide one towards novel design decisions. To this extent, Green recommends "[concentrating] on the 'standard remedies' and their trade-offs" while admitting that this may lead towards awkwardness for which the framework can only provide partial solutions to [17].

3.3.2 Quality Attributes

In this approach [11], languages are evaluated based on how well they achieve various "quality attributes" with respect to the values and goals of the language's audience. These attributes are then grouped into three categories based on what kind of property the attribute imparts on the language and the kinds of methods required to prove such claims. These three categories are:

1. Formal Properties (such as type safety and correctness guarantees)

2. Observational Properties (such as efficiency and compilation time)

3. Effects on Programmers (such as error-proneness, expressiveness, and understandability)

While the first two properties can be determined through mathematical proofs and empirical measurements of language performance, the third category is much harder to make claims for due to its subjectivity and dependence on audience. As such, the framework recommends conducting empirical user studies.

3.3.3 Rhetorical Framework

In their paper "A Rhetorical Framework for Programming Language Evaluation", Muller and Ringler expand on the "quality attributes" approach by adding a fourth category:

4. Aesthetic Properties - The Human-centered aesthetic appeal of the language [31].

Having recognized that each of the categories require different methods for justification—with the "Effects on Programmers" and "Aesthetic Properties" claims[6] being particularly hard to justify due to their dependence on audience—Muller and Ringler developed a rhetorical framework based on argument theory and rhetorical analysis that enables one to make and justify such claims.

In this framework, claims about a language are established through showing what effects the language's features have, and how the results of these effects can satisfy pre-determined metrics for evaluating the language. While these metrics are still subjectively chosen, by using such a framework, one is able to provide enough information about their reasoning (as well as the temporal situation) to allow for others to follow the same reasoning. In addition, one may chose these criteria based on existing work (such as the Cognitive Dimensions framework) to make claims more convincing. As such, while the conclusions from such a framework may be weaker than those derived from an empirical study, it provides a mechanism for evaluating (and justifying) similar conclusions when one lacks the resources to conduct a study with users.

3.4 Designing Programming Languages

Building upon the strategies for evaluating programming languages (discussed in Section 3.3), a few methodologies have been proposed for the process of designing programming languages. This section provides an overview of two such methods. The first approach described (Section 3.4.1) takes an interdisciplinary approach to language design by viewing it as a typical software development project where human-oriented design methods can be used. The second approach described, PLIERS (Section 3.4.2), expands upon the interdisciplinary design approach by detailing a full language design process and rational for using various methods.

3.4.1 Interdisciplinary Programming Language Design

In the Interdisciplinary Programming Language Design approach [11], the design of programming languages is seen as an iterative process where designers cycle through two phases:

1. Requirements and Creation

2. Evaluation

[6] These are sometimes referred to as Category 3 and Category 4 claims respectively

During the Requirements and Creation phase, language designers use methods such as interviews and corpus studies to understand the needs of the language's target audience. Having an idea of what these needs are, the designers then use *natural programming* exercises to understand ways in which programmers may naturally approach solving problems containing these tasks. With this in mind, the designers create a rapid prototype of the language for use in the evaluation phase.

During the evaluation phase, language designers assess their prototype against the various criteria for each quality attribute—a concept this paper introduced. Once this evaluation is complete, the designers can return to the Requirements and Creation design phase to address issues with their prototype until they are satisfied with the results.

3.4.2 PLIERS

PLIERS (Programming Language Iterative Evaluation Refinement System) [12] expands upon the Interdisciplinary Programming Language Approach and provides more detail as to a complete approach to language design. In addition, PLIERS also adapts traditional methods for the specific context of designing programming languages that are targeted at software engineers and "require developers to learn challenging programming concepts or think in a new way". To achieve this, PLIERS breaks down the process of language design into the five phases of need finding, design conception, risk analysis, design refinement, and assessment (as described below) with designers able to return to previous phases if needed.

Phase 1: Need Finding In the need finding phase, researchers assess "what programming problems does the user have, and how might a new programming language help the user achieve their programming goals?" This is typically done through a corpus study, interview, or contextual inquiry [12].

Phase 2: Design Conception Having identified the user's needs, the language designer works to create various theoretical models that may satisfy these requirements along with prototype designs for what each of these languages may look like.

Phase 3: Risk Analysis In this phase, "the designer assesses and prioritizes usability risks in the proposed design." This can use both theoretical methods (such as applying the Cognitive Dimensions framework and comparisons to existing systems) as well as user research to guide decision making.

Phase 4: Design Refinement In the fourth phase of PLIERS, language designers enter an iterative phase of development. During this time, designers refine the language by repeatedly refining the language's theory and prototype based on the results of empirical methods (such as formative usability studies, *natural programming*, and case studies).

Phase 5: Assessment Satisfied with the results of their refined language, the designers conduct a final summative usability study. In contrast to the previous studies which focused on determining if design goals were met, this assessment allows the designers to evaluate how well programmers could use the language to perform realistic tasks.

Through applying this framework, researchers have been able to create two programming languages:

Glacier and Obsidian—both of which significantly deviated from the original language's design based on the feedback that resulted from the study. From this success, the creators of PLIERS believe that it has the potential to be a useful design tool for incorporating human-oriented methods into the design of programming languages in a variety of contexts—including concurrent computing [12].

3.5 Summary

Programming languages arose due to the need for humans to communicate processes to computers in an unambiguous manner that the computer can understand and perform. Given the challenge in instructing computers with individual verbose operations, the design of programming languages has shifted towards human-readability, and, as such, these languages have become a tool that programmers communicate to each other with as well as computers. Designing these languages, however, is a challenging task due to numerous design design decisions that must be made—each of which can have a significant impact on the language. This has led to the development of various frameworks for evaluating languages as well as design methodologies for striking a balance between the language's goals and the user's needs.

4 Critique of Existing Design Methods

Despite the efforts to create programming language design frameworks, few options exist for the low-cost and rapid prototyping of novel languages designed for a specific audience or task. Due to the complex nature of programming languages and our own biases towards familiarity, many of the traditional user-centered formative design methods are only able to capture limited results. For instance, in designing the Obsidian programming language, researchers conducted a study where they asked programmers to propose names for certain keywords that would be unique to the language; however, "results were too inconsistent to justify a particular choice in the language; all suggestions were distinct, and some of them were [unsound proposals]" [12]. Similar problems also occur with other common methods such as *natural programming* as the programmers involved in the studies tended to either propose unsound languages or simply use whatever language was most familiar to them—thus negating the method's ability to discover new ways to conceptualize languages. Due to these limitations, m uch of the formative research in programming language design tends towards using summative methods to make formative decisions. For example, researchers may formulate a design question for a programming language and then conduct user studies to determine which design had the least errors or was the fastest for programmers to use. To limit the burden of such methods, features are usually tested individually by implementing the feature in a similar existing language or by using a similar language as the basis for a Wizard of Oz experiment [12]; however, this may not always be an option for novel languages, and these methods still require a significant amount of time from members of the language's audience who may be hard to contact or recruit for the study.

In contrast, frameworks such as Muller and Ringler's Rhetorical Framework for Programming Language Evaluation avoid this problem by describing how one could make claims about a programming language's effects on users through rhetorical arguments based on existing language design principles and audience-dependent concepts—thus lacking the need for user studies [31]. As a language designer, however, as we need a mechanism to evaluate and compare various languages in order to refine our language, it would be ideal if this process could be flipped so that way, instead of trying to develop explicit claims about a language, we could reveal the impact of the language through studying it within the context of its audience and goals. Then, with our results grounded more in properties inherent to the language's design than our own view of what characteristics stand out, we could use rhetorical arguments to hypothesize about the impact of of these results.

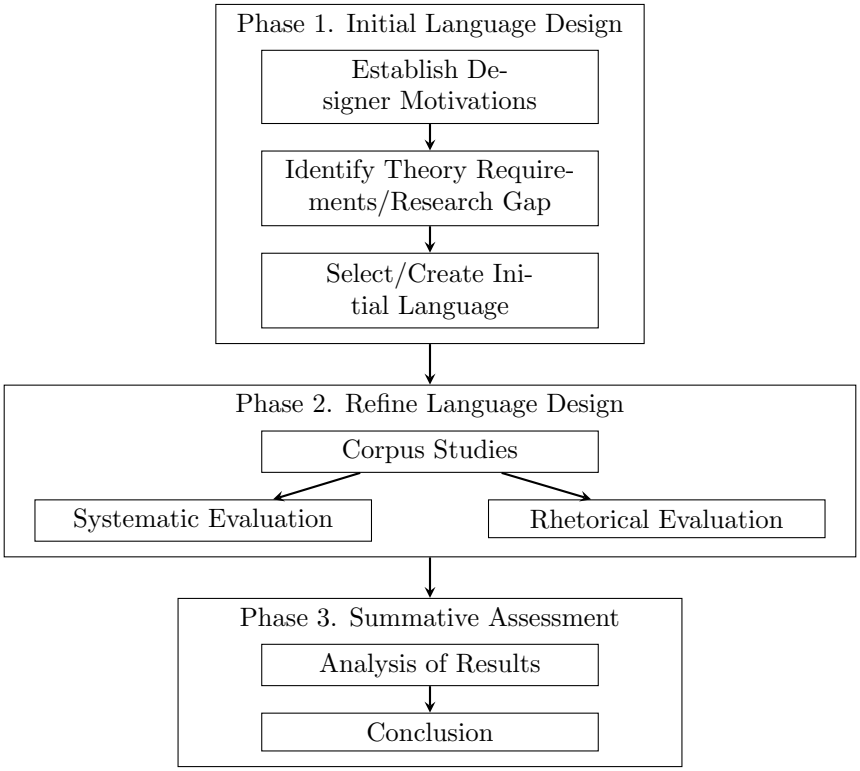
5 A New Language Design Framework

Due to the nature of this project where implementations of similar languages are limited and having limited resources to conduct studies with, for this project to be successful, I developed a new low-cost framework for the early stage design of programming languages intended to overcome the limitations of existing frameworks (as described in Section 4) by answering the question of: "Despite having limited resources, how can we design and evaluate a programming language based on audience needs until we can determine that it would be valuable to apply more intensive methods or that the language is an inadequate solution?"

To develop this framework, I drew inspiration from the structure of the PLIERS framework as it appeared to have a thorough design. However, instead of relying directly on user methods, my framework focuses on uncovering how the language impacts programs written in it within the context of the language's domain (the tasks that the language's audience would expect to be able to do with the language). As a result, this framework places a higher emphasis on demonstrating a language's expressiveness as, not only was that something that existing frameworks struggled with, it is also an important step in determining if a novel approach is sufficient for the problems it seeks to address. As such, the user-oriented considerations in this framework may be less generalizable than frameworks like PLIERS (and they may lack the granularity to answer specific questions—such as what syntax would be the best for the language); however, in return, we are able to more quickly iterate upon a language's and assess if a language should be a candidate for further work. This framework (illustrated in Figure 3) has three major phases:

1. Initial Language Design (Section 5.1)
2. Refine Language Design (Section 5.2)
3. Summative Assessment (Section 5.3)

Figure 3: Language Design Framework



5.1 Initial Language Design

Given a novel task for a programming language to solve, the first phase in my framework is to develop an initial language theory to base the rest of the study on. To accomplish this, the designer first establishes their motivations for the project and what things they care about in designing such a language. This is because, despite having to design a language for a target audience, there are no universal best practices for programming language design [7] (with representations of code becoming more diverse as the need to program enters into more domains [41]). This means that there is no objective way to qualify what features a designer should or should not care about. Instead, there exists a feedback cycle where the designer's views impact the language's audience, and the audience's views impacts the decisions the designer considers.

Having established their motivations, the designer is now in a position to evaluate existing theories and languages to determine a research gap that they believe is worth investigating. This can be done by evaluating the languages from a technical and rhetorical perspective in order to determine their strengths and weaknesses relative to the designer's motivation and intended audience. Throughout this process, the designer is also able to refine their motivations as they become more familiar with the domain—thus providing an initial set of requirements for their research.

Finally, having an initial set of language requirements, the designer needs to establish a starting point for their work. This can be done either through selecting an existing language to work with

or by creating their own language designed to satisfy these requirements. In addition, at this time, the designer may wish to conduct other methods in order to make their subsequent work easier. For instance, if there are certain theoretical properties that they care about, the designer may wish to prove them ahead of time so that way iterating on the language while ensuring these properties continue to hold becomes easier.

5.2 Refine Language Design

Having established a candidate calculus, the language designer needs a mechanism to critique and revise their language from both a systems-oriented and user-oriented and design aspect: we must understand what kinds of things can (and cannot) be expressed in our language and what is the impact of how they are expressed (or their inability to be represented). In this respect, the language designer must recognize their role as a procedural author and push the bounds of their language in order to understand if it, in fact, satisfies the language's technical requirements (as a tool to solve problems) and social requirements as a mechanism for communication.

As this method, however, is designed to prelude intensive research methods by providing a rapid mechanism for language evaluation and revisions (thus maximizing the value of later methods if deemed necessary), in order to achieve this, the language designer first develops a corpus of programming tasks in the language's target domain. Having this, the designer can then write sample programs in their language for each of the programs in the corpus. From a technological perspective, this helps the designer determine the language's expressiveness by forcing them to write a vast array of programs—thus finding potential limitations of the language (for which they may wish to revise their language to address).

Now having sample programs in the new language, the designer is able to rhetorically compare code written in the language with the original samples to determine how their language design impacts the ways that tasks are accomplished and expressed. For this, the designer must consider, for each of the programs, what things do the languages have in common, what things do the languages do differently, why the languages have these similarities and differences, and what the impact of these are. Then, by looking at these results across the samples, the designer can identify how their language would impact the programs written in it. With this, the designer can then use concepts such as the cognitive dimensions framework[7] to describe what effect the language may have on programmers (and giving them a way to discuss if these language differences result in a positive, negative, or neutral impact). This may also help the designer develop alternate representations of the language based on how common patterns could be made easier to utilize. Using this information, the designer can either revise the language to improve it further or move on to Phase 3 of the framework.

5.3 Summative Assessment

Having refined the candidate language, the final phase of my framework is to conduct a summative assessment of the theory to determine if it was a viable candidate for future work, or if it had issues that would require a new theory to solve. To determine this, using the results of the language revision, the designer evaluates the strengths and weaknesses of the language based on their motivations and goals for the language and its audience. From this, the designer then has the opportunity to address how goals may have shifted throughout the research and the goals that they

[7] See Section 3.3.1 and [17]

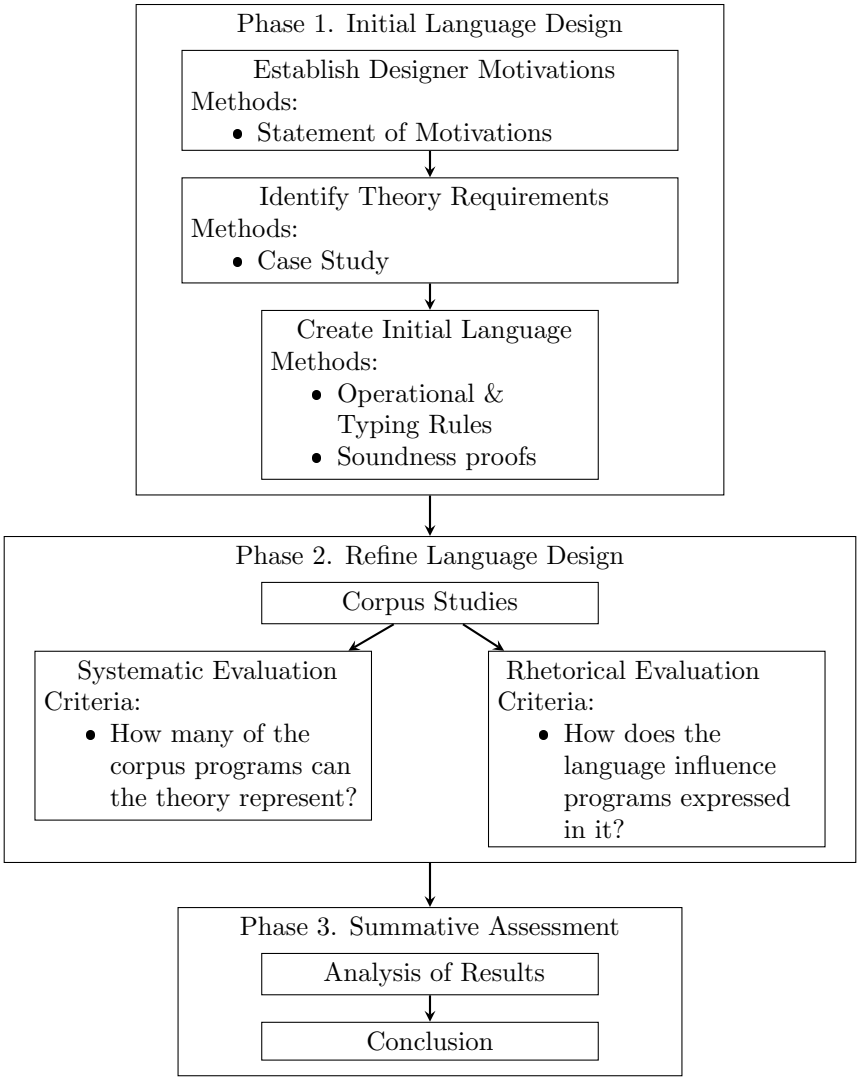
may have been unable to accomplish, and how this impacts the research results and future directions for the project. Finally, based on this, the designer determines if the language's limitations warrant rejecting the language (for instance, because it fails to be expressive enough for its target domain) or if future work is warranted (in which case, what should the future work be[8], and how this work could address the language's limitations).

[8]For instance, should work continue in the development and revision of the language, or is it ready for more intensive methods?

6 Methodology

The goal of this project was to develop a more expressive process calculus that programmers can use to communicate distributed, concurrent, and mobile tasks while retaining correctness guarantees. To accomplish this, I used my framework (Section 5) to design a methodology for this specific language design task as shown in Figure 4.

Figure 4: Project Framework with Corresponding Methods



6.1 Initial Language Design

Due to the limitations of the existing process calculi, I wanted to design my own calculus as a basis for this project (instead of using an existing one). In order to accomplish this, I first needed a mechanism to quantify the strengths and weaknesses of various theories so that way I could develop the requirements for my own language. For this, I conducted a case study as described in Section 6.1.2. Based on these results, I then designed an initial calculus to satisfy these requirements (Section 6.1.3). Finally, before proceeding onto the language refinement phase, I performed a series of correctness proofs in order to ensure that the language would be well behaved and easy to reason about (Section 6.1.4).

6.1.1 Statement of Motivations

Given that the design and evaluation of programming languages is subjective and that this, specific, project approaches this task while not being able to directly communicate with members of the language's target audiences, it is important to establish my own background and motivations for this project as they will bias how I approach this project. Because of this, as suggested by the project's advisors, I adopted the concept of a "positionality statement" [19] from the social sciences into a statement of motivations with the goal of enabling others to both understand the rationale for my decisions and views throughout the project as well as providing them with the ability to assess if my project's conclusion aligns with the stated project goals. As such, in this statement, I describe my background knowledge and experiences that have led to this project, how this influences the way I view this field, and, thus, what my goals/motivations for this project are, and why I have these goals and motivations. In other words, while the claimed motivations, intentions and conclusions of a paper are a performative utterance, by providing the rationale behind the projects, the reader is given the requisite conditions to evaluate if the researcher's claimed conclusions are consistent with their claimed intentions—something which could be proven through a rhetorical argument similar to those described in [31].

6.1.2 Identify Language Requirements

To ensure that the requirements of my own language would result in a novel approach that improves upon the existing alternatives, I conducted a case study on a variety of process calculi. For this, I created a list of calculi that I found during my background research—ensuring that they covered a variety of different approaches. I then developed the categories for the case study by reading the papers which describe each calculus and listing their important features, limitations, and how well they adhered to linguistic best practices (such as use of consistent metaphors)[9]. I then evaluated how well each of the languages supported these features/followed these practices using a three-point value analysis system as shown in Table 1.

[9]For the purpose of the case study, I treated limitations as missing features. For example, if a limitation was that the language was untyped, the case study would evaluate the languages on if they are typed.

Table 1: Value Analysis Point System

Points	Reasoning
1	Language directly supports feature/linguistic best practice
0	Language indirectly supports feature/linguistic best practice (usually in other literature)
-1	Language does not support feature/linguistic practice (or requires extensive modification)

Having completed this evaluation, I was able to identify gaps in current language functionality, trade-offs associated with certain features, and common features shared by process calculi. From this, I developed a set of language requirements for my language based on feature gaps I wanted my language to fulfill and concepts I wanted my language to maintain support for.

6.1.3 Design a New Process Calculus

Having determined what requirements my calculus should have and the benefits/drawbacks of various approaches, I identified the techniques that were most closely related to the goals that I had for my language, and I studied how these techniques worked. I then used these techniques, languages, and theories as a starting place for me to develop my own theory. For this, I primarily wrote typing and operational rules (Sections 6.1.3.1 and 6.1.3.2 respectively) as well as other types of rules when necessary such as invariants (Section 6.1.3.3) and structural equivalences (Section 6.1.3.4).

6.1.3.1 Typing Rules Typing rules allow us to put constraints on a programming language by stating that only certain types are valid for certain operations. For instance, a typing rule for addition may specify that each term being added is a number. Such a rule may look something similar to:

$$\frac{\Gamma \vdash e_1 : \mathsf{number} \qquad \Gamma \vdash e_2 : \mathsf{number}}{\Gamma \vdash e_1 + e_2 : \mathsf{number}}$$

Where $\Gamma \vdash e_1 : \mathsf{number}$ and $\Gamma \vdash e_2 : \mathsf{number}$ are the rule's premises (conditions which must be met to draw the rule's conclusion), and the rule's conclusion is $\Gamma \vdash e_1 + e_2 : \mathsf{number}$. In this case, the rule's premises specify that, in typing context Γ, given two expressions in the language (e_1 and e_2) that each have a type of number, $e_1 + e_2$ is a valid expression that, similarly, is of type number.

As such rules allow us to reason about a program without having to run it, by robustly defining the typing rules for each operation in a language, we are able to provide certain guarantees about how valid programs can (or cannot) function [18].

6.1.3.2 Operational Semantics The operational rules for a language allow us to determine how a language steps from one expression to another—thus describing how a program runs [6]. For instance, an operational rule for addition could be written as:

$$v_1 + v_2 \mapsto v_3 \text{ (where } v_3 = v_1 + v_2)$$

6.1.3.3 Invariants Invariants are properties that must hold true across given operation(s).

6.1.3.4 Structural Equivalences Structural equivalences allow us to define how terms of a certain structure have an equivalent functionality as terms of another structure. For example, given a number n, we could stipulate that $n \equiv -(-n)$ as a means to describe how the the double negation of a term is equivalent to the original term. Such a rule then allows us to substitute $-(-n)$ in a proof for n.

6.1.4 Prove Language Correctness

Having designed an initial language theory, I proved my language's correctness through progress (Section 6.1.4.1) and preservation (Section 6.1.4.2) proofs. Not only are these two proofs critical for having a well-functioning language, by proving them, I was able to establish that my language was type-safe. This means that, given a program, it is possible to determine if it is well-formed and well-behaved without needing to execute it [18].

6.1.4.1 Progress Proving progress requires that we show that, in a language, the following holds true:

$$\text{If } e : \tau \text{ then either } e \texttt{ val} \text{ or there exists } e' \text{ such that } e \mapsto e'$$

Where e is an expression in the language, and τ is its type. Proving this in a language is critical as it means that any expression in the language will either evaluate to a value or get stuck in a loop performing computation: it is impossible for any expression to get stuck as all expressions are either values or performing some computation.

6.1.4.2 Preservation Proving preservation requires that we show that, in a language, the following holds true:

$$\text{If } e : \tau \text{ and } e \mapsto e', \text{ then } e' : \tau$$

Where e and e' are expressions in the language, τ is their type. Proving this is a basic requirement for proving type system in a language as it allows us to demonstrate the types behave in a predictable manner that can be determined without running the program.

6.2 Refine Language Design

Having established a candidate calculus for our novel problem, I needed to determine if the calculus was expressive enough to allow developers to communicate tasks in the problem domain with the theory. To accomplish this, I began a process of iteratively evaluating and refining my language. To achieve this, I first developed a corpus of programming language tasks that would be common use cases that my language would need to handle (Section 6.2.1). Then, by writing these sample programs in my language (Section 6.2.2), I could identify it's limitations within the greater societal context of how members of the target audience use programming languages. In addition, by rhetorically evaluating these sample programs, would be able to determine how my language's design impact the programs written in the language (Section 6.2.3). Having identified

these limitations and design impacts, I could then revise my language by repeating the language design process (starting with the steps in Section 6.1) with my current language as a starting point.

6.2.1 Corpus Study Data Collection

To develop my corpus of language tasks, I conducted background research to identify common design patterns and use cases for distributed and concurrent programs (Table 2 Paradigms section). For this, I looked at course materials and textbooks that specialized in distributed and/or concurrent programs and identified commonly used designs, concepts, and example programs/assignments used to teach these concepts.

I also researched what kinds of distributed and concurrent algorithms and applications currently exist (Table 2 Common Tasks section). As many of these programs had complex elements beyond their functioning as a distributed or concurrent program, for such programs, I focused on ensuring that my language could describe the kinds of communication used by these applications as well as its ability to represent various design patterns that such software development projects would employ.

In addition, having been working on an implementation for my programming language for fun in my free time, I was able to supplement the corpus study with a variety of test cases (Table 2 Test Cases section) that I had previously written which, being inspired by test cases I had written for a C-style language, allowed me to notice further differences between my language and existing programming languages.

Table 2: Corpus Study Program List

Paradigms		
Name	Appendix	Source
Message Passing†	–	[42]
Client-Server Model	D.2	[21, 26]
Peer-to-peer Model	D.3	[42]
Publish/Subscribe Model	D.3	[26]
Remote Procedure Call (RPC)	D.4	[26]
Distributed Objects‡	–	[42]
Object Spaces‡	–	[42, 26]
Mobile Agents	D.5	[21, 26]
Common Tasks		
Name	Appendix	Source
Quicksort	D.6	–
Reliable Data Transmission (RDT)	D.7	[24]
Test Cases		
Name	Appendix	
IsPrime	D.8	

† Inherent property to the language due to the type system and is thus demonstrated in all samples.
‡ As Bismuth Calculus is not object oriented, these paradigms were omitted from the case study. Future work could consider if/how such concepts could be implemented by finding a mechanism to develop objects within Bismuth Calculus by using syntactic sugar.

6.2.2 Systematic Evaluation

To evaluate how expressive my language was from an algorithmic perspective, I needed to determine how many of the programs in the corpus could be represented by programs in my language that would achieve the same effect. For this, I studied each of the programs in the corpus to understand what it would accomplish when run. Using this knowledge, I then wrote pseudocode in my language (substituting in traditional language features when required and allowed based off my language's definition[10])to accomplish the same task. To ensure that I had completed this step correctly, I proved each of my programs by using my typing rules to ensure that it would type check, and my operational rules to demonstrate that the program would behave in the desired manner (Appendix D). Having repeated this step for each program in the corpus, I evaluated what percentage of programs my language was able to represent and why my languages was unable to represent the programs that I failed to translate. Based on this, I then evaluated if I needed to revise the language to address these limitations or if I should proceed with the rhetorical evaluation.

[10]For example, closures or other features that could be used in a manner that would violate linearity were not allowed, but features such as if statements whose functionality could be approximated through existing features were.

6.2.3 Rhetorical Evaluation

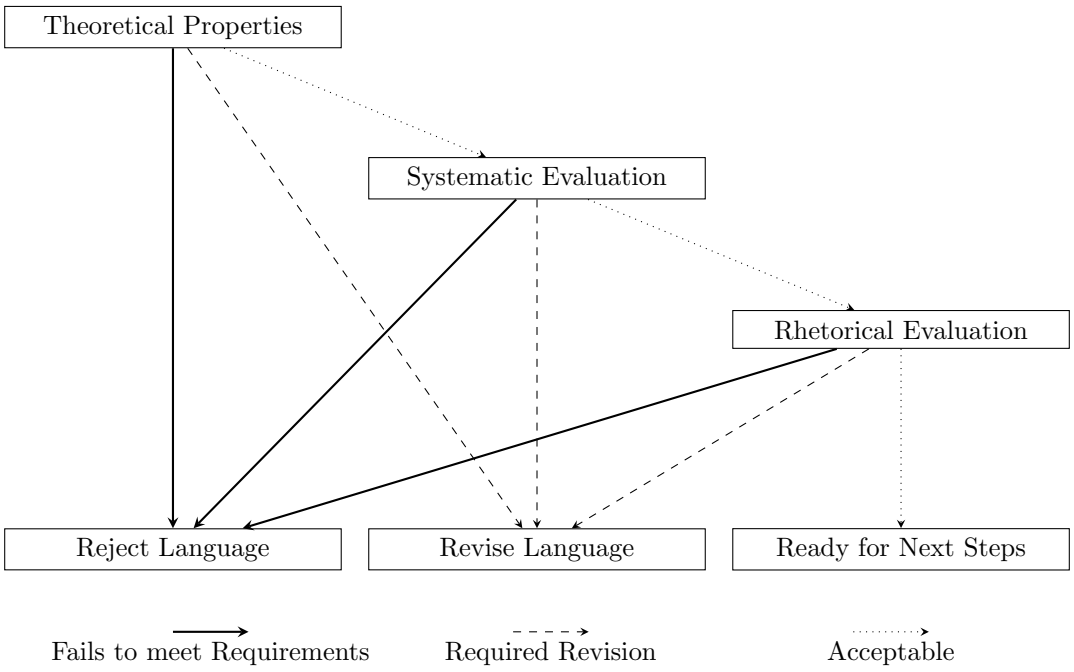
Having a collection of programs written in my programming language that corresponded to languages in my corpus (for which I had sample code for in existing programming languages), I compared and contrasted how each programming language expressed the same concept. For each of these similarities and differences, I then explained why my programming language differed from the existing language or why the two representations were the same, and what impact each of these decisions had on the language. In doing so, I hoped to identify what concepts each language highlighted and concealed (thus acting as a textual silence), how the languages conveyed information, and how the procedural rhetoric of the languages differed. From this, the language could then be revised based on the project's goals and audience as well as the cognitive dimensions framework and our understanding of how programmers comprehend code.

To interpret these results, I applied methods inspired by the Grounded Theory Method (GTM) to synthesize my initial rhetorical comparisons (written as a list of stream of consciousness comments comparing the implementations) into a description of how the language impacts the programs written in it. I decided to use this method as GTM is known for its ability to take raw data (such as what I had) and use it to develop, test, and critique theories about phenomena within the data—even when such connections may not be at first apparent[30].

6.3 Summative Assessment

For the language's summative assessment, I had to evaluate how well my programming language met the language's goals as established during Phase 1 (or explain how and why those goals had shifted). In the case of this project, that meant evaluating the language's theoretical properties and expressiveness. For the theoretical properties, this was evaluated based on if I had been able to successfully prove all of the properties that I had set out for the language to have (Section 6.1.4). To evaluate the language's expressiveness, as with the Phase 2 refinements, I judged the language based on two criteria: its ability to represent tasks and how it expressed tasks. For the former, I wanted to be able to represent the vast majority of the corpus programs in my language (while understanding why I could not represent the other programs, what modifications it would take to represent them, what the impact of these changes would be, and if such changes would be reasonable). For the latter expressiveness criterion, I scrutinized the similarities and differences that my programming language has, and put together an argument weighing the strengths and weaknesses of my language relative to the other languages. As shown in Figure 5, these three criteria create a decision tree where the language is rejected if it fails to meet any of the above criteria, the language is revised further if so much as one criteria demands it, and, otherwise, if all criteria agree that the language is ready for further steps, then do we proceed in that direction.

Figure 5: Framework Conclusion Standards



7 Results

This section details the work process and results of this project. The section starts by describing the initial language design phase—including the statement of motivations and requirements gathering potions of the project (Section 7.1). Then, it moves on to discussing the the language's evaluation (Section 7.2). Finally, the section finally closes by discussing the revisions made to the based on the evaluation's findings, remaining language limitations, and correctness proofs (Section 7.3).

7.1 Initial Language Design

To establish how I approached this project why I approached it in the manner that I did, the first phase of my framework involved me writing a statement of motivations for the project (Section 7.1.1). Having established this, I conducted a case study (described in Section 7.1.2; data in Appendix B) to identify limitations of existing calculi as well as common traits between calculi that I wanted to preserve. Based on this, I then developed a set of requirements for my language (Section 7.1.2). Finally, with these requirements in mind, I created a new calculus (Section 7.1.4) for which my later evaluations would be based on.

7.1.1 Personal Motivations and Background

My main interest for this project stemmed from having programmed a variety of projects that either required me to use threads in order to speed up computations, or to coordinate between several processes (both in cases where the processes where on the same device and in cases where they were distributed across several). In developing these applications, I consistently found myself having to re-write code to make it work with various methods of concurrency, manually parse data sent between devices, and debug complex logical problems caused by asynchronous communication and processing as well as a lack of a shared memory to work from. Recognizing that being able to perform distributed and concurrent computations is critical to a wide variety of applications from web services to computer graphics, artificial intelligence, scientific research and aviation (where the traffic collision avoidance system is responsible for ensuring safe distances remain between aircraft), when I first learned about the field of process calculi, I found it to be a fascinating area of study that may be able to help people write these kinds of applications more easily. Because of this, despite having little background in computer science theory, I started to research the various calculi that existed. However, in doing so, I found that many of these theories were hard to learn and work with. As such, I started this project in hopes of finding an easier way for people to communicate these kinds of programs which, to me, meant building a new language as that would be an easier place for me to start than using an existing language. Based on my limited background at the time, I knew that I would want this language to:

1. Have types that describe communication protocols and variables so that way we can prove correctness and more easily reason about programs.

2. Treat all variables, language features, and metaphors as consistently as possible. For example, this would mean the calculus should be higher-order as it would be inconsistent to have limitations that would prevent this.

3. Treat everything as a process (similarly to how everything is a function in λ-calculus) so that way any task could be easily distributed without having to restructure the code.

4. Not rely on having a shared global state as we may not have shared memory between processes in certain methods of distribution.

5. Clearly represent what is being communicated over each channel so that way we can understand where resources are located.

7.1.2 Case Study Findings

Having evaluated seven different process calculi (Table 3), I found that two of the seven (π-calculus and Ambient calculus) were untyped—which limits our ability to reason about the runtime behavior and correctness of a program statically. At the same time, however, the four languages which supported typed communications (Lolliproc, πDILL, GV, and Functional GV) often used notations that require a strong theory background to understand and had challenges representing complex communication patterns.

For instance, the four session type based languages required considerable effort to establish two-way communication or protocols which include repetition. In this respect, they are much better at representing parallel functions than how process evolve over time through collaborative communication. At the same time, despite languages such as π-calculus and Ambient calculus being able to describe this long-term style of interaction between processes, they are only able to represent repetition in their communication protocols through establishing infinitely persistent programs which are always present to handle any loop or repetition. Such a restriction, thus, makes it incredibly challenging to model a system where a program may repeat part of its protocol for an arbitrarily long duration of time before proceeding onto other operations.

Table 3: Case Study Languages

Language	**Approach**	**Source**
π-calculus	Traditional π-calculus language.	[37]
Ambient calculus	Extends π-calculus with ambients.	[10]
Lolliproc	Uses session types and pass-by-value.	[28]
πDILL	Extends π-calculus with session types and *DILL*.	[9]
GV	Asynchronous session types with linear types	[15]
Functional GV	Functional language extended with session types.	[25]
Distributed Join-Calculus	Extends Join-Calculus with locations.	[14]

Another weakness of many of the process calculi was that, with the exception of the Distributed Join-Calculus, none were fault-tolerant. However, while this is an important feature, due to its added complexity and time constraints, I decided to leave this as a consideration for future work.

Beyond this, while many of the languages supported comparable features, many of their flaws dealt with how they communicate what computations programs perform. For instance, while all of the calculi except for π-calculus adopt the notion that channels should only link communication

between two processes at a time—a necessary requirement to ensure unambiguous processing[11]—only four of the six languages with this property utilized a global state for channels—something which is unrealistic [14] and requires additional effort to ensure that channel names are always unique. In addition, many of these languages express sending and receiving data in a limited or inconsistent manner with languages generally falling into one of three categories:

1. Treat all variables linearly—thus limiting our ability to create certain data structures (as described in Section 2.5.2).
2. Treat communicating across channels as linear, but make a copy of any non-linear resource sent across a channel so both sides retain it.
3. Use a dual intuitionistic logic system so that way we can consistently model linear and non-linear resources.

Out of all the languages evaluated, only πDILL used a dual intuitionistic logic system—thus providing it with an ability to describe the location and use of resources more consistently than any other language evaluated.

Despite these limitations, many of the calculi set good precedence for what kinds of features a process calculus should have. For instance, the four session typed calculi[12] and Distributed Join-Calculus highlight the benefit of being able to concisely represent short-term function-like operations function-like operations (where one value is exchanged for another) as not every interaction requires long-term two-way communication. Other precedents include many of the calculi allowing for mobility: a program can be sent from one process to another. In many respects, this a very natural way to extend the idea of first-class functions (from λ-calculus) into process calculus. Similarly, all of the calculi treat channels as higher-order and thus allow for a channel to be sent over another channel. Not only does this help establish more complex communication patterns by enabling us to link processes in otherwise challenging (if not impossible) ways, it also allows us to implement features such as remote referencing regardless of if values are communicated linearly or non-linearly (as copies) via channels.

7.1.3 Initial Requirements

Based on the results of the case study, I identified three areas of improvement for this work to focus on that, when put together, provide a useful set of features which no other calculi appears to have:

1. Easily describes complex typed communication between processes
2. Consistently modeling the location and communication of resources.
3. Describing communication without the use of global state or persistence.

To meet this first requirement, my language should be able to model any communication protocol that a regular language can describe—including two way and repetitive communication. As a downside, this meant that I had to be prepared to trade deadlock freedom for expression; however,

[11] If we allowed more processes to communicate over a channel, it becomes incredibly challenging to ensure that communication protocols are followed.

[12] Lolliproc, πDILL, GV, and Functional GV.

given there are many interesting programs that may be impossible to prove deadlock free (despite operationally having this property) and that there would likely be ways of restricting the language to obtain this guarantee, it was a tradeoff that I was willing to make. While, ultimately, the language resulting from this research is deadlock free[13], I conducted my work under the assumption that deadlock freedom would be up to the developer to enforce and that the language would be responsible for proving correctness in communications: if communication occurs between processes, it will do so without protocol violations.

For the second requirement, my language must be unambiguous and consistent with its description of the location and communication of resources while also not compromising on language functionality. For instance, if channels must be communicated linearly, then all communications should be done linearly; however, this is does not mean that the language should treat all resources linearly due to the inherit limitations of linear types.

Finally, to fulfill my third requirement, the language should describe communication without relying on some sort of global state: channels must be stored locally in processes thus making it clear exactly who is able to communicate across that channel. In addition, while processes may run indefinitely, they are expressly prohibited from existing persistently: processes may loop for an infinitely long time, however, their existence must be seen as temporary and, thus, they must describe how they would complete their protocol and terminate—regardless of if these operations ever occur. This requirement would also help my language function compositionally with external processes with whom we may only have their channel and not their source code to execute. While this, when combined with our lack of persistence, means that such a language would need a way to handle the case where we attempt to connect to a resource which has since terminated or continued past their repetitive offer to serve clients, such fault tolerance will be reserved as the subject of future work.

In addition to these areas of improvement, I also wanted my language to preserve the following properties that many of the existing calculi possess:

4. Allow for higher-order channels and program mobility.
5. Have channels link exactly two processes.
6. Easily model short-term (function like interactions)[14].

Through meeting these requirements, my language should preserve many features of existing calculi while making strategic trade-offs which allow it to describe processes in a new way.

7.1.4 New Process Calculus

The first language requirement that I went to address was easily allowing complex typed communication between processes. For this, I decided to use a modified version of sequent calculus to represent protocols in my language. My rationale for using sequent calculus stemmed from the fact that it has analogs for: sending data, receiving data, repetition as chosen by the local process, repetition as chosen by the other process, internal choice, external choice, parallel operations and

[13]This is not necessarily true in the extended future version of the language used in the Corpus Studies where we allowed for parallel protocols and for both processes to use these protocols in a conflicting sequential manner.

[14]Due to my second requirement, long-term interactions should be described similarly to short-term ones, and both should be reasonably easy.

sequential operations. While these operations combine to enable us to model complex regular communication patterns (and allow us to represent repetition without persistence), because sequent calculus proofs rely on deriving a contradiction through using the law of the excluded middle (i.e., proving $A \vee \neg A$), many of the connectives are defined as the dual of each other. This becomes a problem in achieving my language's third requirement: describing communication without the use of global state or persistence. Traditionally, process calculi rely on a global state of channels to keep the various processes synchronized and enabling them to make a distinction between the sending and receiving channel for each pair of communicating processes. This limits the composability of programs in such languages as we need to type check all programs at once instead of each individually.

As this calculus will not have a global state, operate synchronously, or make such a distinction between ends of a channel, we instead need a way to dualize a protocol so that way they can correctly cooperate independently. If we were to take this approach using regular sequent calculus, we would occasionally find ourselves dualizing the connectives in ways that change the program's meaning—such as converting sequential steps on one process to parallel operations on the other. To address this, in my modified version of sequent calculus, I introduce two new constructs, ; and |, to replace $\otimes$ and $\parr$ respectively—each of which are the dual of themselves. ; represents one action followed by another[15]. Similarly, simultaneously would represent two actions that can occur in parallel (and thus separately)[16]. However, due to time limitations, the parallel channel operator would be omitted from this initial language definition.

To meet my second language requirement of consistently modeling the location and communication of resources, inspired by ambients and πDILL, I decided to take a similar dual intuitionistic approach. Such a system allows us to maintain a consistent type system that reconciles the requirement for channels to act linearly while still allowing other resources to act non-linearly. For this to work, we simply require that only linear resources are capable of being sent across a channel. And, if we have a non-linear value that we wish to communicate, then we are required to make a linear copy of it. This copy can either be a regular linear type which must continue to always act linearly, or it could be a persistent linear type. In this latter case, this means that we can use the linear resource an unlimited number of times; however, in this state, the resource is essentially read-only as, each time we use the resource, we are presented with another copy of it. This, however, is a fairly minor limitation as, having unlimited copies of the resource means that we must be able to convert it back into a non-linear type. As such, this system provides us with an explanation for why channels must always remain linear (they are not persistent) while other resources can shift between being linear and non-linear. Thus, not only can we consistently describe resources and their communication in such a system, we also clearly communicate where these resources are located due to the fact that, any communication across a channel represents moving a resource (or a copy of it) from one place to another. However, for sake of simplicity, the version of the calculus in this paper only considers channels to be linear resources, and all other resources are considered non-linear.

As my modified sequent calculus system for protocols addresses my language's first and third requirement, and using dual intuitionistic linear logic addresses my language's second requirement, this leaves us with requirements 4–6 which ensure that the language preserves some of the good features that the other process calculi possess. Of these, requirement four is fulfilled through treating channels as linear values and programs as values (these can be either linear or non-linear).

[15] In this respect $A \multimap B$ would be defined as $A; \neg B$, and $A \otimes B$ would be defined as $A; B$.

[16] In this respect, $A \parr B$ would be defined as A, B in my version of linear logic.

Requirement five is fulfilled through channels being linear, and the definition of executing a program only providing a channel to the parent and child process—thus making it impossible for a channel to link anything but two processes. Finally, problem six is addressed through the use of sequent calculus where function-like invocations are just simple protocols. This resulting baseline calculus (which was used as the basis for the later corpus studies) can be found in Appendix C.

7.2 Iterative Language Evaluation

Having completed the first phase of my language design framework (described in Section 5.1, adapted to this project in Section 6.1, and results in Section 7.1), I moved onto the iterative refinement phase of my framework (described in Sections 5.2 and 6.2). For this, I first wrote versions of each program in the corpus in my own language (Programs in Appendix D; Baseline language definition in Appendix C). I then evaluated how many of these programs were possible to express in my language, and why certain programs could not be expressed in Section 7.2.1. Following this, I conducted a rhetorical evaluation that compared the programs written in my language to how they would be expressed in existing languages in order to determine how my language impacts the programs written in it (Section 7.2.2). The main findings from these evaluations were then summarized in Section 7.2.3.

7.2.1 Systematic Evaluation

Of the seven programs in my corpus study, my language was able to represent two of the programs without any modifications or simplifications (Quicksort and IsPrime), three of the programs with simplifications (Client-Server model, Remote Procedure Call system, and Mobile Agents), and limited versions of two programs with modifications and simplifications (Publish/Subscribe model and RDT).

Across these tasks, the main factors limiting the implementation of programs appeared to be the complexity of protocols and data management required. The two programs that my language represented without modifications (Quicksort and IsPrime) are both algorithms that only interact with processes that have a simple function-like protocol: they are provided with data, and they return a single result without data having to be shared simultaneously between processes. As such, these implementations closely resemble their counterparts in traditional programming languages. As the complexity of the task increased, however, we quickly discover limitations of the language which require us to make simplifications to our program if we wish to proceed. This was first apparent with the need to represent unreliable channels where the connection—a concept needed for all of the other programs (assuming they would have to communicate across the internet or another unreliable medium, which is a realistic assumption to make). This tends to be a limitation of many process calculi that rely on session types as they have a requirement called *protocol fidelity* that requires processes to fulfill their protocols. As there is currently no way to represent such cases in my language, if we consider this to be a problem for future work, paradigms such as the Client-Server model, Remote Procedure Calls, and Mobile Agents become possible.

Even with these simplifications, however, certain types of programs remain impossible in the language. The two cases of this in my corpus study were that of RDT and the Publish/Subscribe model. For the RDT program, this was because, in order to guarantee the reliable transmission of data across an unreliable medium, we may have to send the same resource across the channel multiple times. As such, we must be able to use any resource that we wish to reliably send in a non-linear nature. This becomes a problem if we ever wanted to send a linear resource reliably—such

as a channel. And, while such an algorithm could be written in a manner that respects linearity (in the sense that we could ensure that the linear resource would only exist and be used in one place despite having to potentially send it multiple times), it would be impossible for the language to understand such a system as being anything but a violation of linearity. After all, RDT algorithms are not provable correctness guarantees but instead probabilistic functions which only work given enough communications and a reliable enough transmission medium.

In contrast, the Publish/Subscribe model is impossible to a lacking of features within the language. The most prominent of such features is the ability for a channel to contain multiple subprotocols that are all running in parallel. While, in other tasks, a similar concept can be represented through passing multiple channels to a single process for it to use, the problem here is that we need:

1. Two processes to be able to interact with each other in multiple ways at the same time, and, currently, each process is limited to having one channel—meaning we would need several subprocesses that just forward data in order to achieve a similar effect

2. A way to be able to split linear resources between parallel operations while also being able to share persistent linear resources over an implicit for of mutex.

While such a concept seems reasonable, future work would be required to investigate how this could be done, and how feasible it is.

In addition, both of these programs suggests that the language needs a a mechanism for bringing ! protocols into while loops, and to be able to conditionally access a limited number of elements provided by an ! protocol and leaving the remaining elements (if there are any left) to another part of the program to handle—a concept sometimes seen in the corpus study as the `acceptWhile` operator[17].

7.2.2 Rhetorical Evaluation

Despite the limitations of my language identified through the systematic evaluation, I decided to proceed with the rhetorical evaluation on all of the programs in the corpus. This was because, even in the simplified version of the programs written in a version of my language that was augmented by additional hypothetical features, I thought that the resulting comparisons would still yield useful data—especially as many of the limitations thus far could be addressed through future work.

Based on applying Grounded Method Theory to my notes from the Rhetorical Evaluation (Appendix D), I hypothesize that the primary challenge of using my language is due to having to write programs that the language's rules are able to prove uphold correctness guarantees. The impact of this can be viewed through its direct impact on code in the language (Section 7.2.2.2), as a consequence of what metaphors and concepts are embedded within the language (Section 7.2.2.3), and as a question of how to represent new concepts in a programming language while still working within the larger context of programming language design (Section 7.2.2.4).

7.2.2.1 Background Information for Results Traditionally, human languages push back against the concept that they are governed by a fixed set of rules [27]. Instead, language tends to evolve over time—incorporating changing social contexts and new needs for expression [41]. Even

[17] In the Publish/Subscribe Helper, this is simplified to be a two parameter version of the `accept` rule which will loop no more than the number of times specified by the second parameter.

when concepts and phrases fall out of favor, they still exist as parts of the language through record of their use and thus influence future vernacular.

Programming languages, however, are much more fixed. Not only are they limited through having to eventually correspond to a fixed set of instructions understood by the computer, they are also limited by their design—including what theoretical principles they uphold. Despite these apparent limitations, programming languages are capable of being universal models of computation: they are able to evaluate any possible computation—including the simulation of any other programming language. In addition, once a programming language has been instantiated, they tend to cope with changing contexts and needs through libraries which, instead of adding to the language's definition, establish metaphors within the language for expressing these concepts in a modular manner which can be reused throughout other projects.

This suggests that, programming languages are able to overcome their limitations through leveraging procedural rhetoric to provide users with the metaphors needed to capture a wide array of emergent behavior for which the original designers could never have foreseen. However, despite layers of abstraction, there eventually becomes a point at which the dissonance between the language's rules and the developer's conceptualization and ideal expression of a problem breaks down. As all language is based on metaphors, once this occurs it can become incredibly challenging for developers to devise a manner to communicate within the confines of the language as their task has gone from communicating an algorithmic computation to establishing that there is a way to achieve such a concept within the language's rules. In this translation, however, it is almost inevitable that the meaning of the program will have to change as one is forced to achieve the same ends through a different means.

One manner in which this was particularly apparent in my language was through the need to uphold the property known as preservation. Preservation stipulates that if a program is in a valid state which evaluates to another state, then the resulting state must also be valid. Without this property, programs may act unpredictably and fail in unexpected ways. For example, in a language without preservation, it would be possible to have a valid program that adds together three given numbers; however, in evaluating this program, we may find that after adding the first two numbers together, we are no longer in a state where we can add the third number to our value.

In languages such as mine, preservation becomes a critical requirement as, it enables us to guarantees that programs will correctly and completely utilize all channels as stipulated by the channel's protocol—a concept known as *protocol fidelity*. This property is a major factor in allowing us to offload basic and repetitive tasks (such as data parsing and *serialization*) to the compiler—thus enabling the developer to focus on more novel and interesting tasks. In addition, by establishing what kinds of data that a program can expect to receive, the language may be able to perform these mundane tasks more securely, as we can limit the need to share potentially unsafe data such as programs as, knowing what programs are possible to receive, we could ensure that we have those programs accessible locally. In contrast, without this property, then there is no point in using protocols to begin with as the language is unable to guarantee that the protocol's meaning will carry any weight. These unexpected failures would also mean that it is impossible for our language to guarantee that linear resources (those which must be used exactly once and thus are only located in one place at a time) would be used properly. This would cause a cascading failure in which all programs in the entire system fail as the impact of each successive failure results in the violation of these properties across more programs.

7.2.2.2 Proving Correctness Despite the need for programs to uphold correctness guarantees to be considered valid in my language, because the language can only contain a finite set of rules for determining if a given program has these correctness properties[18], programs in my language sometimes require developers to take additional steps to prove correctness despite the program, logically, being correct. Not only does this require extra work by the programmer and change the program's meaning, but it can also be dangerous as it can:

1. Introduce untestable (and potentially incorrect) code into a program which may later become live due to a change in protocol or through being copied into other live portions of the code due to another developer mistaking it as live tested code which could solve problems elsewhere in the codebase.

2. Require us to write more complicated (and potentially less efficient) code that is more challenging to understand and work with.

An example of this first case can be observed in my language's Publish/Subscribe helper program where the loop on lines 18–21 can clearly be observed as unreachable. As shown in Figure 6, this is because the only way for us to get to this code is if `envOpt` is storing a value of type `Unit`, and, from lines 10–14, we can tell that this is only possible if `rest` loops for zero times. Nonetheless, our language has no means to understand this, and requires us to add in the additional code to ensure that the loop is correctly processed.

Figure 6: Publish/Subscribe Helper - Lines 10–21

```
10      ((Channel<!+Message> | Channel<?-Message>) + Unit) envOpt = Unit;
12      accept(rest, 1) {
13          envOpt = rest.recv();
14      }
15
16      match envOpt
17          | Unit u => {
18                  accept(rest) {
19                      (b: Channel<!+Message> | r: Channel<?-Message>) = rest.recv();
20                      link(b, r);
21                  }
```

An example of the second case can be found in the flawed IsPrime example, (as illustrated in Figure 7) where the language is unable to determine that the only possible time that the channel c can be used within the while loop, is on line 6 where the program will then exit—thus preventing any protocol violations. This limitation requires us to engineer a mechanism to break out of the loop so that way the language can verify that the protocol is used correctly (as seen in the corrected IsPrime example). Similarly, in the RDT sender program, the requirement to uphold *protocol fidelity* forced us to use *spinlock*s, a notably inefficient mechanism, to handle timeouts as we cannot execute two processes and only care about the first one to finish—even if both processes are providing us with a non-linear resource that could be discarded. *Protocol fidelity* also restricted the RDT program to only function with non-linear data as a resource may have to be sent multiple times to unsure reliable transmission from one device to another, and while it may be possible to craft such a system

[18] in contrast to a human interpreting the language who could apply new proof techniques to determine if a program still upholds these guarantees.

in a manner which does not result in any correctness issues, it would be impossible to prove such an algorithm with the language's rules.

Figure 7: IsPrime - Correct (left) vs Flawed (right) Comparison

```
1  define isPrime :: c : Channel<+int;-boolean> = {
2      int n = c.recv();
3      int i = 3;
4
5      boolean done = false;
6      boolean ans = true;
7
8      while (!done && i < n)
       {
9          if (n / i * i = n)
10         {
11             done = true;
12             ans = false;
13         }
14         else
15         {
16             i := i + 2;
17         }
18     }
19
20     c.send(ans);
   }
```

```
5      while (i < n) {
6          if (n / i * i = n) { c.send(false) exit; }
7          i := i + 2;
8      }
9      c.send(true);
```

Proving program correctness to the language also becomes a challenge when a program needs to recognize a protocol that is logically equivalent to the one provided, but that the language cannot understand as equivalent. This is because there may exist multiple ways to translate between two logically equivalent programs. Thus, even if the language could understand them as being equivalent, it would be unable to determine how the code was intended to function. For instance, given two separate channels of $!\rho$, we should be able to join them together into one channel; however, this could be interpreted in various different ways including: alternating between the two channels, using the entirety of one channel before moving onto the second, and using whichever channel current is able to provide us with a protocol to follow. To resolve such discrepancies, one is currently forced to write intermediate programs that handle these conversions in real time through accepting channels as inputs, and then having their remaining protocol describe the resulting transformation (see Figure 8). While this gives us the ability to transform channels in an incredibly complex manner and the ability to describe and reason about how programs will interact, because this process requires the introduction of intermediate programs to perform translations, these kinds of operations can be computationally intensive and, in some cases, excessive as even the most straightforward transformations that could be determined at compile time (or handled synchronously by the process that needs the channel transformation) must be run separately as their own process.

Figure 8: Sequential Channel Merge Program

```
define SequentialMerge :: c : Channel<+Channel<!ρ>;+Channel<!ρ>;?not(ρ)> = {
    Channel<!ρ> a = c.recv();
    Channel<!ρ> b = c.recv();

    accept(a) {
        more(c)
        link(a, c)
    }

    accept(b) {
        more(c)
        link(b, c)
    }

    weaken(c)
}
```


This limitation becomes particularly apparent in the Mobile Agents program when we need to provide local resources to the mobile program despite it being agnostic to the device where it is being executed. Traditionally, if one wanted to move a program from one system to another during execution, they would use a system similar to the one shown in the Java Mobile Agents program wherein the program would be serialized and sent to another device which would deserialize the program and execute it. When this occurs, the program would be automatically granted access to all local resources that it imported as if it were on the original device (or it would crash if its current host is missing these resources).

In my language, however, there is no notion of global resources as they would introduce the possibility for *race conditions* and complicate our requirement to uphold linearity as per our need to ensure preservation. Instead, each program is only has access to the resources local to it, and it can only obtain resources from the outside world through its channel. As such, to share local resources across programs, we would have to perform such channel splitting operations on each resource to ensure that we retain a copy to share with other programs. In some respects, this is not necessary a bad thing as it means that we have to document how programs interact with resources and we are given more flexibility over what kind of access we provide programs with. At the same time, however, this can become increasingly complicated and tedious to manage as we are presented with various types of programs which each require different resources—such as the extreme case of the Permissive Mobile Agent Server where we have no idea what resources any given program needs. In addition, any time we make a change that requires providing different resources to a program, we would have to rewrite a significant amount of code to enable this—making debugging techniques such as adding print statements an arduous (if not impractical) task.

One potential work around to this would be to have a linear collection of persistent channels (those of type $\uparrow ?\rho \uparrow$) which the language can split into copies which are automatically passed to programs similarly to how, in object oriented languages, a `this` pointer is automatically passed to member functions of an object. Various mechanisms, such as subtyping or the automatic weakening of unused persistent resources, could then be introduced to ensure that linearity is preserved even when the agent only uses a subset of its provided resources. If we are able to do this, however, we then run into the question of if our program can now have global variables so long as they are persistent and, if so, how do we manage resource splitting, do mobile processes use the mobile

resources of their originating device or their receiving device? If it is the originating device, how would this work if communication was over an unreliable medium without violating preservation? If it is on the local device, and how do we ensure that the device has all the necessary resources for the program to function? In addition, we have to balance the trade-off of being able to provide specific resources to specific programs versus the benefits of granting every process access to whatever it needs (much like how existing languages import resources).

7.2.2.3 Richness of Protocols From a linguistics perspective, preservation is analogous to the felicity condition of a performative which allows us to view a program as a set of true statements instead of performative utterances. This is because, in proving that preservation holds for a program, we have proven that every utterance in the program can be successfully processed. This is something often taken for granted in type safe languages to the point where we frequently view every line of code as statement of what will occur as opposed to a performative whose truth will ultimately be tested at runtime.

With session typed languages, this is highlighted through *protocol fidelity* as not only does it describe what operations our program needs to perform over time to be considered valid, it also describes how we can expect the programs that we are interacting with to function. As such, our programs are thus bound by the metaphors used to dictate the meaning of a protocol; and, when these metaphors fail to capture complexity of the task we wish to model, such programs become impossible to express.

Throughout the corpus study, these kinds of limitations forced me to simplify five of the programs[19], primarily due to these programs corresponding to tasks which required communicating over the internet as such communication channels can unexpected close or abort early as per the decision of either process—something which my language lacks the ability to represent without a protocol violation. While this is a significant limitation for this version of the calculus, proceeding with the corpus study despite this revealed how, even with these simplifications, programs became much more challenging to write as the mapping between the real-world metaphors we use to understand problems and the metaphors in the language started to break down—something which could be described as a violation of Green's Closeness of Mapping concept [17].

Even when writing the comparatively simple Client/Server program in my language this quickly became apparent as I spent a relatively significant amount of time debating if the server should follow a ? protocol or a ! protocol. Initially, the former case would seem to make the most sense as we may be used to putting code that reads from a server socket into a while loop that runs as long as the server wishes to read from the socket. However, then the client would connect to the server through an ! channel which could mean that the client would connect to a server and somehow negotiate the ability to gain entry into the protocol loop (or handle the case where the loop is already closed). In contrast, if we have the server running an ! protocol, then the clients would see a channel of type ?—which can be easily split among various programs for them to make various requests. This would mean that the server no longer has control over how long it provides its service for, if we have to concede that only one program can control the loop's duration, then this option is a much easier to model.

Compounding this problem is that, at first, it may seem like it should be possible to write a protocol transformation which allows programs to abort early by transforming channels of form

[19]These programs were: the Client-Server model, Remote Procedure Call system, Mobile Agents, Publish/Subscribe model, and RDT.

$\rho_1; \rho_2; \rho_3...$ into:

$$\mathsf{ExternalChoice} <\mathsf{Error}, \rho_1; \mathsf{ExternalChoice} <\mathsf{Error}, \rho_2; \mathsf{ExternalChoice} <\mathsf{Error}, \rho_3; ...>>>$$

via an error-transform program (similar to the error monad). By performing such a transformation on both ends of the communication system in a manner where our intermediary process is given the internal choice between sending an Error whenever there is a disconnect or the continuation to the remaining portion of the protocol whenever data has been received correctly. While such a transform would require us to implement a feature where a protocol could be composed of multiple parallel subprotocols, the need for such a feature was already demonstrated by the Publish/Subscribe program. Unfortunately, this program also illustrate several problems with this proposed solution:

1. As such a transform can only apply to the protocol, if any of the steps in the protocol involved sending or receiving another channel, then the sent or received channel would not have such a transform applied to it—thus somehow providing reliable data transmission via an unreliable channel which is a contradiction[20].

2. Even if we could somehow extend such a transform to the data sent by such a channel, we would have to engineer a mechanism to ensure that all channels close at once when their parent channel is closed—which would be incredibly challenging (especially as channels may be forwarded across many devices) if not impossible.

With this in mind, it may seem theoretically impossible for such a language to handle unreliable channels while maintaining preservation. However, if preservation is simply the felicity condition for the performance established by our protocol, then if we could introduce a new protocol (or channel modifier) which represents the possibility for an abort by either process at any step in the computation, then we may be able to solve such a problem. While such channels may be more challenging to interact with due to the need to ensure linear resources are handled correctly, they would allow us to draw a nice comparison between the dual intuitionistic nature of our language wherein we have analogous features for non-linear and linear resources. In this case, unreliable protocols would be non-linear as resources sent across them could be discarded[21], reliable channels would be their linear equivalent, and channels with the abort structure would help us bridge between linear and non-linear.

While introducing such a protocol may make most uses of the error transform program obsolete, considering such a transformation also reveals something interesting about the rhetoric of the language: there is a class of protocols (which includes that recognized by the error transform program) for which, despite being able to write a correct and potentially useful program that recognizes them, the language lacks a notation for the protocol. This can occur when a program recognizes a protocol that is both polymorphic and recursive—thus, the true protocol can only be fully expanded by other programs which connect to these programs as only they would have the knowledge of the data sent to the polymorphic program.

This makes writing such a program impossible for the sole reason that we cannot write the protocol recognized by the program in its definition—which could be avoided by not making this

[20]While this may sound analogous to TCP or other methods of RDT which provide a reliable channel over an unreliable medium, the difference here is that the TCP channel itself can fail if the underlying medium closes or becomes too unreliable. In contrast, the example with the transform program would suggest that higher-order channel sent over the unreliable channel can never fail—even when the underlying channel does.

[21]They would also have to be structured in the form of $?+\tau$, $?-\tau$, $!+\tau$, or $!-\tau$

a requirement and using protocol inference to determine the protocol of a program only when it is needed. This, however, limits our ability to document (and understand) what a program does as we would lose a critical piece of information from the program's signature. This would also prevent us from using the typing system as a a safeguard to ensure that the program follows the protocol that we want it to. Instead, we would have to rely on inspecting the protocol of such programs by writing various client programs in an attempt to test the protocol of the program would unfold to what we expect it to. This would, however, be as challenging as testing a program as the protocol's type would be determined by a Turing complete system—thus making it possible for various edge cases to exist in the protocol's definition. Even if we were to overlook this problem, we would still find that it causes challenges if we ever wanted to use such programs in a higher-order manner as, being unable to represent the program's protocol, we would have to pass such programs by name exclusively. This would cause higher-order use of such programs to be limited to a defined set of programs instead of subtyping.

While this may seem like a good reason for us to expand the possible notations for protocols, the only way that they could be made as expressive as the base language is if they too were made Turing complete—meaning, our options are to either limit our language's expressiveness or to make the language so expressive that we have to accept that there may be cases where traditional notations fail us, or that we need to have multiple redundant definitions of the same thing: a Turing-complete protocol definition that matches a Turing-complete program's recognition of the protocol. Expanding the language's syntax would also only further complicate an already complex system and add more challenges to editing existing code.

7.2.2.4 Writing Code Beyond the question of what metaphors a language has and what concepts these metaphors allow programmers to express, in designing a language, one must determine how these concepts are represented. While this may seem like a simple task given that we have already have a representation of the language from our proofs, many times, it is possible to have concepts in the theory which do not show up in the surface level language that people would interact with and vice versa. Often these concepts are either needed by the theory to prove correctness (and giving a user control over them would break these proofs) or as an alternate representation of concepts derivable in the theory that is easier for users to communicate with (a concept known as *syntactic sugar*). In addition, even when concepts are shared between the theory and the language, they frequently have different representations due to each having different practicality constraints and audiences.

When considering the notation for protocols in my language, I needed to find ways to communicate seven new protocol operators—many of which have mathematical notations that either use symbols in ways that conflict with with the symbol's traditional use in programming languages, or use symbols that would be impractical to include in a programming language. For example, programming languages typically use the | operator to represent an or operation. While such a metaphor would seem to make sense to use for branching protocols in my language, because we have to make a distinction between an internal choice (where we get to choose what protocol the system to follow) and an external choice (where the remote system has to decide what protocol we should follow), using the | symbol would not make sense as it would be unclear which of the two concepts it is representing. In addition, we cannot the traditional mathematical notation external and internal choice of & and $\oplus$ in our programming language because & would be too easily conflated with the and operator and $\oplus$ does not appear on standard keyboards[22].

[22]It would, likely, also require the use of special file encodings.

Because of this, throughout my corpus study I replaced protocols of form $\rho_1 \oplus \rho_2$ and $\rho_1 \,\&\, \rho_2$ with InternalChoice<ρ_1, ρ_2> and ExternalChoice<ρ_1, ρ_2> respectively as this notation would communicate what each of these operations do, and they follow a syntax which is similar to that of generics which—existing in other languages—would hopefully make it easier to infer what these protocols mean. As a downside, however, not only do these representations take more characters to express, they also rely on commas to separate the protocol options which may be easy to overlook as the complexity of ρ_1 or ρ_2 increases—suggesting that this syntax lacks good TS knowledge [35] (albeit, formatting each protocol option on its own line may help with this [17]).

Despite not using the | operator for internal or external choices, such an operator would still likely be needed to represent a logical or operations. This raises the question of if the symbol can still be used to represent other channel concepts despite them being less well connected to the metaphor of an "or". For example, | is frequently used in theories to represent parallel execution; however, using | for both "or" and parallel execution depending on if it is describing a channel or a value could make its meaning ambiguous due to its context-dependent nature. Another example of this ambiguity in my language arises with the ! operator as it is typically used to represent a not or negation; however, in linear logic (one of the concepts my language is based on), it is the "Of Course" operator which represents a loop whose duration is controlled by another program. As the need for looping protocols appeared quite frequently in programs, I decided to keep using this representation in the initial language as it was compact and, within the context of protocols, easy to work with. This decision, however, meant that I would need to introduce a new operator to negate a protocol. While this was represented using $\neg$ in the theory, being an impractical symbol for use in the surface level language, I ended up replacing it with the not function in the corpus study; however, this raises the question of if this syntax should only be used for channels, or if (for consistency) we should also use it for boolean resources (despite the existing conventions and ease of use that the ! syntax provides such as allowing the not equals operation to be represented by !=).

Even these modified representations, however, are far from perfect as it is possible to write channels that are hard to read despite not being incredibly complex in nature. While this is in part due to the language's syntax making it, at times, hard to distinguish between protocol elements (hindering our ability to parse and understand the underlying logic) much of this challenge arises due to the lack of a secondary notation for protocols—making them convey much less information about the pragmatics of a program than their function parameter counterparts. In a function's signature, each parameter is is given a name that usually provides some information regarding how that variable will be used by the program. This allows other programmers to get a better idea of what a function might do and how to interact with it without having to read the documentation or implementation for the function. In contrast, the current representation of protocols in my language does not contain such a notion and, thus, the meaning of a protocol is much more ambiguous. In addition, given that channels evolve over time and are not strictly receive operations, finding a secondary notation for protocols to use to convey additional information about what each step does is a complicated problem worth further consideration.

This challenge is compounded by the fact that sometimes the metaphors used to represent protocols can be misleading. While I found this was generally less of an issue when reading existing protocols, it was particularly evident when writing them. This is because there can be some differences between the way that we think about concepts in natural language and how we may represent the same concepts using the language's protocols. In some respects, this is similar to how language may make one conflate "and" and "or" when learning boolean logic—especially when

dealing with negations. For example, if we have a process where we are given the option to perform a task or move on to a different task, and, every time we complete the current task, we must make the decision of if we continue to repeat the current task or move onto the next one, it may seem natural that this process would be described with an internal choice. This, however, is incorrect; instead, we would simply be given a loop that we control the duration of. Then, whenever we decide that the loop is over, we step to the next task.

Despite these challenges, however, once we have a channel for a program, we are able to learn a lot about what kind of structure the program needs in order to recognize the channel. This can make writing programs much easier so long as the programmer is able to accurately track the state of the channel's protocol as it changes throughout the completion of various operations—something which is currently not highlighted by the language's syntax. This ability to guide a programmer through an implementation also comes at the cost of making changes to a program much more challenging if the protocol that a program interacts with changes. While this, to some extent, is true with traditional languages whenever a function signature changes, because protocols define what a program must do (instead of just the data that a function requires, and the type of data that it returns), the impact of these kinds of changes is much greater in my language—thus leading it to have a higher "viscosity" [17].

Even with this guidance, writing programs that correctly recognize protocols may be particularly challenging for people learning the language as, while they might be used to tracking the state of variables through mutations in code, the ways through which one interacts with channels are inconsistent. In the original language, `send`, `receive`, and `case` (external choice) are all managed through a syntax that looks like we are invoking a function that is a member of our protocol; however, `more` and `weaken` appear as if we are sending the protocol to a function. There is, however, no particular reason for this distinction and either notation could be considered preferable depending on if one views the code from the perspective of it being an object oriented or functional programming language (of which, it is neither). The one potential exception to this is that of `case` as, traditionally, control flow operators follow the latter syntax. While the other control flow operators, the `accept` and `while` loops, do use this syntax, the semantics of the two loops differ in ways that seem inconsistent as the `while` loop operates on a boolean expression whereas `accept` operates on a channel of form $!\rho$. This is because the while loop is designed to help us use $?\rho$ protocols which, having control over their use, we are able to bring them into multiple loops. In contrast, as we do not have control over how many times a $!\rho$ protocol will loop, we are required to focus on one at a time and completely exhaust its use before proceeding with the next step in our program. Finally, one of the most challenging syntactic elements is that of the internal choice. In the original language, an internal choice is handled through a syntax that appears like an array access on the channel where the specified index selects which of the protocols to follow. For example, if c : Channel<InternalChoice<ρ_1, ρ_2>>, then $c[1]$ would make c : Channel<ρ_1> and, similarly, $c[2]$ would make c : Channel<ρ_2>. Such a syntax can be hard to use as it requires determining which protocol is at each index, and it is much easier to overlook than the other channel control elements (or mistake its function for another language feature).

7.2.3 Evaluation Summary

Based on these evaluations, my language appeared to be a fairly promising theory as it was able to express many of the concepts in the corpus study relatively easily with the language primarily abstracting away tedious tasks—such as data parsing—leaving the programmer to focus on the more

interesting (and application specific) implementation work. Despite this, however, I also identified the following five limitations whose impact is significant enough on the kinds of tasks likely needed by the language's audience to prevent the language from being practical in its current state:

1. Enabling ! and ? protocols to be usable within nested loops.
2. Implementing a protocol that allows for aborting communication.
3. Implementing parallel operations within protocols
4. Determining a better notation that is more consistent and more human-friendly
5. Developing optimizations for protocol transforms and resource management

Despite these limitations, given the overall success of the theory and the reasonable potential for the limitations to be successfully addressed through future work, I believe that it makes the most sense for the language to enter another round of revisions and re-evaluations prior to being deemed ready for more intensive methods or being deemed an inadequate solution. In fact, based on these findings, I revised the language to address the first limitation (Section 7.3). While this revision also made some progress towards the fourth limitation, it, along with the the other limitations, were primarily left for future work (as detailed in Section 8.1).

7.3 Refined Calculus Definition

Based on the results of the initial evaluation, I developed a revised version of my language (Appendix F) to address the problems it identified with nested loops. In the initial theory, bringing a ? channel into both while loops and accept loops could cause issues if one were to weaken the channel in the loop as, despite usually being a valid operation, it would cause the loop to break on subsequent iterations as we would no longer have the channel to interact with [23]. Similarly, the evaluation identified the need to bring protocols of form ! into looping control flow structures without causing analogous issues whereby recognizing the ! protocol in a loop, the channel would no longer be of form ! for subsequent loops. As described in Section 7.3.1, I was able to resolve these problems and prove soundness in the resulting language. While this does add some complexity to the language, Section 7.3.2 describes how much of this can be accomplished without impacting the user level language that programmers would interact with, and how the language could be presented to the user in various ways to make it easier to work with. Finally, the section closes with a brief discussion on the revised language and how it fits in with the project's initial goals (Section 7.3.3).

7.3.1 Revised Theory

As seen in Figure 9, as with before, in the revised version of the language, we have types for channels, programs and the typical non-closure base types one would expect in a programming language.

[23] While one potential solution to this was to have protocols of form $?\rho$ become ρ when they entered a loop (and thus preventing them from being weakened), from the corpus study, it became apparent that such a functionality would be limiting.

Figure 9: Bismuth Calculus Refined Syntax

$\tau ::=$	*Types*	
	$\uparrow\rho\uparrow$	Channel Type
	ρ	Program Type
	τ_b	Base Type
$\rho ::=$	*Protocols*	
	$+\tau;\rho$	Receive
	$-\tau;\rho$	Send
	$?\rho;\rho$	Why Not
	$!\rho;\rho$	Of Course
	$\rho \,\&\, \rho$	External Choice
	$\rho \oplus \rho$	Internal Choice
	$^\bullet\rho$	Guarded Resource
	1	Unit Protocol
$m ::=$	*Protocol Messages*	
	v	Value
	$\mathsf{SEL}[\mathsf{a}]$	Select/Project
	$\mathsf{START{-}LOOP}$	Start Loop
	$\mathsf{END{-}LOOP}[!\rho]$	Exit Loop
$a ::=$	*Projection Annotation*	
	$\cdot \,\&\, \rho_2$	Left Projection
	$\rho_1 \,\&\, \cdot$	Right Projection

$P, Q ::=$	*Program*	
	e	Expression
	$P; Q$	Sequence
	$c\langle e\rangle$	Send
	$c(v)$	Receive
	$\texttt{while}(e)\{P\}$	While Loop
	$c.\texttt{case}(P,\ Q)$	External Choice
	$c[i]$	Internal Choice
	$\texttt{more}(c)$	Unfold
	$\texttt{weaken}(c)$	Weaken
	$\texttt{accept}(c)\{P\}$	Accept Loop
	$\texttt{acceptWhile}(c,\ e)\{P\}$	Accept While Loop
	$c = \texttt{exec}\ P$	Execute Program
	$\texttt{exit}$	Exit Program
	$\texttt{skip}$	End Loop Iteration
$e ::=$	*Expressions*	
	x	Variable
	e_b	Base Expression
$v ::=$	*Value*	
	$c :: \uparrow\rho\uparrow : P$	Program
	$\uparrow\rho\uparrow$	Channel
	v_b	Base Value

As with most process calculi, Bismuth Calculus contains rules for sending and receiving data over a channel, offering an external choice between protocol, selecting a protocol to follow from an internal choice, and executing programs[24] as shown in Figure 10 where Γ is the non-linear context and Δ is the linear context of channels. Notably, all channel operations are asynchronous and defined in a manner where we can take the dual of any protocol to obtain the protocol for the other end of the communication to follow. This system enables us to describe incredibly complicated interactions between multiple processes in a more compositional manner than session types traditionally can while remaining deadlock free.

[24] While not shown explicitly in the theory, Bismuth Calculus does allow for higher-order programs.

Figure 10: Bismuth Calculus Basic Typing Rules

$$\text{(TSend)}\quad \frac{\Gamma;\Delta \vdash e:\tau \quad \Gamma;\Delta\{e\}, c:\restriction\rho\restriction \vdash P}{\Gamma;\Delta, c:\restriction\neg\tau;\rho\restriction \vdash c\langle e\rangle.P}$$

$$\text{(TRecv)}\quad \frac{\Gamma\langle x\rangle;\Delta\langle x\rangle, c:\restriction\rho\restriction \vdash P}{\Gamma;\Delta, c:\restriction+\tau;\rho\restriction \vdash c(x).P}$$

$$\text{(TCase)}\quad \frac{\Gamma;\Delta, c:\restriction\rho_1\restriction \vdash P.R \quad \Gamma;\Delta, c:\restriction\rho_2\restriction \vdash Q.R}{\Gamma;\Delta, c:\restriction\rho_1 \,\&\, \rho_2\restriction \vdash c.\texttt{case}(P,\ Q).R}$$

$$\text{(TSelect1)}\quad \frac{\Gamma;\Delta, c:\restriction\rho_1\restriction \vdash P}{\Gamma;\Delta, c:\restriction\rho_1 \oplus \rho_2\restriction \vdash c[1].P}$$

$$\text{(TSelect2)}\quad \frac{\Gamma;\Delta, c:\restriction\rho_2\restriction \vdash P}{\Gamma;\Delta, c:\restriction\rho_1 \oplus \rho_2\restriction \vdash c[2].P}$$

$$\text{(TExec-Prog)}\quad \frac{\Gamma;\Delta, c:\restriction\neg\rho\restriction \vdash Q \quad \cdot; n:\restriction\rho\restriction \vdash P}{\Gamma;\Delta \vdash (c = \texttt{exec}\ \ P :: n:\rho).Q}$$

Note: The $\Delta\{e\}$ and $(\Gamma;\Delta)\langle x\rangle$ operators (as used in TSend and TRecv) represent the removal of linear resources through their use in e and the insertion of resource x into its proper context (depending on if it is linear or nonlinear) respectively. Both of these are formally defined in Appendix F.1.

To help enable this, Bismuth Calculus uses a system similar to subexponentials which enables us to interact safely with multiple sessions within loops—something that many existing calculi do not allow. As shown in Figure 11, this works by marking all existing linear resources as guarded (represented by $^\bullet$ prefixing the resource) prior to entering into a loop[25]. This allows us to distinguish between linear resources introduced in the loop (which must be utilized fully in each iteration) and the linear resources that come from outside of the loop whose use is now restricted to ensure that the loop will be able to complete future iterations successfully.

Figure 11: Bismuth Calculus While and Accept Typing Rules

$$\text{(TWhile)}\quad \frac{\Gamma;\cdot \vdash e:\mathsf{bool} \quad \Gamma;\Delta_1 \vdash P \quad \Gamma;\Delta \vdash Q}{\Gamma;\Delta \vdash \texttt{while}(e)\{P\}.Q} \quad {}^{1}$$

$$\text{(TAccept)}\quad \frac{\Gamma;\Delta_1, c:\restriction\rho_1\restriction \vdash P \quad \Gamma;\Delta, c:\restriction\rho_2\restriction \vdash Q}{\Gamma;\Delta, c:\restriction!\rho_1;\rho_2\restriction \vdash \texttt{accept}(c)\{P\}.Q} \quad {}^{2}$$

[1] $(\Delta_1 \equiv \{x_i : \restriction^\bullet\rho_i\restriction \text{ for all } x_i \in \Delta\})$
[2] $(\Delta_1 \equiv \{x_i : \restriction^\bullet\rho_i\restriction \text{ for all } x_i \in \Delta\})$

To this extent, there are only two kinds of local resources with whom we can interact with while guarded: those of form $?\rho$ and those of form $!\rho$. In the case of $^\bullet?\rho$, this is achieved through the T?Unfold Guarded rule as shown in Figure 12.

[25] Note: $^{\bullet\bullet}\rho \equiv {}^\bullet\rho$ as, currently, there is no use for being able to determine how many times a resource as been guarded beyond knowing if it is or is not guarded.

Figure 12: Bismuth Calculus Unfold Typing Rule

$$\text{(T?Unfold Guarded)} \quad \frac{\Gamma; \Delta, c : \restriction \rho_1; \bullet?\rho_1; \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \bullet?\rho_1; \rho_2 \restriction \vdash \texttt{more}(c).P}$$

Essentially, recognizing that protocols of form $?\rho$ allow us to utilize ρ any number of times, given a protocol of form $\bullet?\rho$, T?Unfold Guarded allows us to obtain an unguarded copy of ρ—thus making the channel of form $\rho; \bullet?\rho$. As with all unguarded resources within the loop, the first ρ of this resulting channel must be recognized prior to the end of each iteration. This ensures that the channel will be available for future loop iterations and that its state at the start and end of each iteration will be the same.

In contrast, using a $\bullet!\rho$ protocol is slightly more complicated as another program controls how many times we have to use the ρ resource To overcome this, one must use the TAcceptWhile-Guard rule as shown in Figure 13.

Figure 13: Bismuth Calculus Accept While Typing Rule for Guarded Protocols

$$\text{(TAcceptWhile Guarded)} \quad \frac{\Gamma; \cdot \vdash e : \mathsf{bool} \quad \Gamma; \Delta_1, c : \restriction \rho_1 \restriction \vdash P \quad \Gamma; \Delta, c : \restriction \bullet!\rho_1; \rho_2 \restriction \vdash Q}{\Gamma; \Delta, c : \restriction \bullet!\rho_1; \rho_2 \restriction \vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q} \quad [1]$$

[1] $(\Delta_1 \equiv \{x_i : \restriction \bullet \rho_i \restriction \text{ for all } x_i \in \Delta\})$

Unlike the typical TAccept rule, TAcceptWhile does not serve as a mechanism for recognizing an ! protocol. Instead, it can run a variable number of times so long as we have more resources from the ! to use. As such, there is no problem using it on guarded protocols because it will always leave the ! protocol intact with the only difference being how many subsequent iterations of the ! protocol are possible.

With the introduction of guarded resources, we now need a mechanism to allow loops to terminate if the only linear resources they contain are guarded. For this I introduced the `skip` directive as shown below in Figure 14.

Figure 14: Bismuth Calculus Skip Rules

$$\text{(TSkip)} \quad \frac{}{\Gamma; \Delta \vdash \texttt{skip}} \quad [1]$$

$$\text{(OPSkip-Continue)} \quad (G; L_i; \texttt{skip}.P) \mapsto (G; L_i; P)$$

$$\text{(OPSkip-Exit)} \quad (G; L_i; \texttt{skip}) \mapsto (G; L_i; \texttt{exit})$$

[1] (For all $x \in \Delta$,$\Delta(x) = \restriction \bullet \rho \restriction$)

In many respects, `skip` can be seen as the middle ground between the `exit` directive which allows a process to finish executing so long as all of its linear resources have been used and the process equivalent of the bottom type wherein it would not matter if we used all of our linear resources as the program would loop forever prior to their use. In fact, the only difference between `skip` and the bottom process is the side condition of TSkip and the presence of the OPSkip-Continue operational

rule. This distinction, however, is critical as, if it were not for the side condition on TSkip, loops could terminate with channels of form $\rho_1; {}^\bullet\rho_2$ which would then result in a protocol violation as, on the next iteration of the loop, the local program would expect the channel to be of form ${}^\bullet\rho_2$ while the remote program would expect $\neg(\rho_1; {}^\bullet\rho_2)$.

To prove that the language was sound with these changes, I completed proofs for both progress and preservation (Appendices G and H respectively). While these both follow the standard inductive proof by cases strategy for proving these kinds of properties, in order to complete the proof, several concepts were introduced that do not otherwise exist at the language's theory level. The first of which is the system's configuration:

$$M \equiv (G; \overrightarrow{L}; \overrightarrow{P})$$

Where G is the global state of the computation[26] and $\overrightarrow{P}$ is a vector of all processes in the system. Each $P_i \in \overrightarrow{P}$ corresponds to a unique $L_i \in \overrightarrow{L}$ where L_i describes the local state of the program. This is described through the rule TypeL which is used by the OK rule which typechecks an entire configuration as shown in Figure 15. Also shown in this figure is the OPUniversalStep rule which is used to allow an individual program step to advance the entire state of the system.

Figure 15: Bismuth Calculus Global Typing & Operational Rules

$$\text{(TypeL)} \quad \frac{\cdot \vdash L_i(x) : \Gamma_i(x) \quad \Delta_i(x) = \Sigma_B^G(\Sigma_O)(c)}{\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)} \quad {}^{1}$$

$$\text{(OK)} \quad \frac{\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i) \quad (\Gamma_i; \Delta_i) \vdash P_i}{\cdot \vdash (G; \overrightarrow{L}; \overrightarrow{P})\ \mathsf{ok}}$$

$$\text{(OPUniversalStep)} \quad \frac{(G; L_i; P_i) \mapsto (G'; L_i'; P_i')}{(G; \overrightarrow{L}; \overrightarrow{P}) \mapsto (G'; \overrightarrow{L'}; \overrightarrow{P'})} \quad {}^{2}$$

[1] (For all $x \in \mathsf{dom}(\Gamma)$ and for all x,c such that $L_i(x) = \uparrow c \uparrow$)

[2] $\begin{pmatrix} \overrightarrow{L'} \equiv \overrightarrow{L}[i \mapsto L_i'] \\ \overrightarrow{P'} \equiv \overrightarrow{P}[i \mapsto P_i'] \end{pmatrix}$

In order to work, these rules rely on two additional concepts: the buffer function (Σ_B) and the protocol oracle (Σ_O). The need for these concepts arises because communication in the language is asynchronous, and at the operational level, channels are split into two queues: one for each process to read messages from. Thus, these functions allow us to reconstruct protocol information from the message buffers and compare them to the expected state of the system (as shown in Figure 16).

[26] This is not necessary shared between all processes, but, instead, representative of an external view of the system used to track the state of our communications channels. This allows us to abstract away the transport of data in the theory, and, in the real world, it corresponds to the notion that data will be transmitted between processes reliably—regardless of how much of the system is aware of this or how the data is transmitted.

Figure 16: Bismuth Calculus Buffer Function

$$\Sigma_B^G(\rho)(c) = \begin{cases} \rho & \text{if } G(c) \equiv \epsilon \\ +\tau; \Sigma_B^{G[c \mapsto q']}(\rho)(c) & \text{if } G(c) \equiv v[\tau]; q' \\ (\Sigma_B^{G[c \mapsto q']}(\rho)(c)) \,\&\, B & \text{if } G(c) \equiv \mathsf{SEL}[\cdot \,\&\, B]; q' \\ A \,\&\, (\Sigma_B^{G[c \mapsto q']}(\rho)(c)) & \text{if } G(c) \equiv \mathsf{SEL}[A \,\&\, \cdot]; q' \\ \text{let } \alpha; !\alpha; \beta = \Sigma_B^{G[c \mapsto q']}(\rho)(c) \text{ in } !\alpha; \beta & \mathsf{START{-}LOOP}; q' \\ !A; \Sigma_B^{G[c \mapsto q']}(\rho)(c) & \text{if } G(c) \equiv \mathsf{END{-}LOOP}[!A] \end{cases}$$

Thus, by providing Σ_B with a message queue and Σ_O (which tracks the expected state of future channel messages), we are able to show how all channels in a system evolve in manners which are consistent with how their corresponding processes expect—enabling us to prove preservation.

7.3.2 User-Level Language

While many of these changes may seem to add complexities to the language that would make it harder to work with, as many of these concepts only matter to the theory, the surface level language remains relatively unchanged despite now enabling new forms of expression. This is illustrated in Figure 17 which demonstrates a more user-friendly version of the syntax where only the syntax elements that one would encounter actually using the language are displayed.

Figure 17: Bismuth Calculus Refined User-Level Syntax

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

Of these remaining elements, syntactic sugar can be used to further improve one's experience

using the language as shown through the examples below.

Syntactic Sugar for Internal/External Choice In the language's typing rules for Internal and External Choice, we are not allowed to have any protocols following the choice. This, however, can be made possible through introducing the following structural equivalents as syntactic sugar:

$$\begin{aligned} \uparrow(\rho_1 \oplus \rho_2); \rho_3 \uparrow \quad &\equiv \quad \uparrow(\rho_1; \rho_3) \oplus (\rho_2; \rho_3) \uparrow \\ \uparrow(\rho_1 \,\&\, \rho_2); \rho_3 \uparrow \quad &\equiv \quad \uparrow(\rho_1; \rho_3) \,\&\, (\rho_2; \rho_3) \uparrow \end{aligned}$$

This allows our language to understand such concepts by transforming what the user writes into something that can be processed by our typing rules. For the user, this helps them write programs more easily as, following either branch in a choice, they may wish the program to follow the same logic and, having such syntactic sugar would enable them to write this once following the branch instead of twice in either branch case. This also helps make the typing rule more consistent with the operational rule for these cases which allows for more code to follow outside of an internal or external choice.

Alternative Syntax for Internal & External Choice Programs One of the challenges identified by the corpus study (Section 7.2.2.4) was that of Internal Choice requiring the programmer to use the index of the desired protocol in order to select which one the programs should follow. One potential alternative to this would be for the user to specify the protocol they wish to follow instead. For example, given the channel c : Channel<InternalChoice<+int, +bool>>, we could allow them to specify c[+int] or c[+bool] to select following the protocol of +int or +bool respectively. While this could become tedious if we had long complicated protocols, we could potentially allow for named choices wherein each protocol offered in the choice is given a unique name. This would also have the added benefit that we could have multiple protocols of the same type in the choice, but, by having different names, we can infer a semantic meaning from them. For example, we could have the channel c : Channel<InternalChoice<getRed : +int, getGreen : +int, getBlue : +int>> where, despite having the choice between three identical protocols, we can understand a difference between what each would do and select between them accordingly through using c[getRed], c[getGreen], and c[getBlue].

Another concern raised in the corpus study was that of external choices appearing more as a function call on a protocol than a control flow operator. To address this, one potential option would be for us to introduce a syntax for it similarly to how pattern matching works in other languages. For example, instead of having to write:

```
// c : Channel<ExternalChoice<A; B>>
c.case(
    //Program to recognize A,
    //Program to recognize B
)
```

We could instead use the following syntax:

```
// c : Channel<ExternalChoice<A; B>>
offer c as
```

```
3     | A => //Program to recognize A
4     | B => //Program to recognize B
```

As an added benefit, by implementing both of these changes, we would be able to decouple the ordering of the protocols in a choice from the code for recognizing the choice protocol—thus making it easier to modify the protocol without having to modify the code. And, in the few cases where the code needs to be modified, we would be more likely presented with a compiler error warning us of this than we would otherwise.

7.3.3 Revised Calculus Discussion

While the result of this revision only fully addresses one of the language's limitations (enabling the use of ! and ? protocols within nested loops), and provides a brief discussion of potential notational improvements, the revised calculus pushes this project forwards by demonstrating how it is possible to construct a sound calculus capable of expressing such complex interactions between channels, and that it is possible to do so while considering common programming language metaphors and audience needs. As such, despite having limitations that prevent this calculus from fully realizing the project's initial goals, I believe this project opens the door for a wide variety of interesting subject for future work (Section 8.1) that has the potential to successfully address these limitations.

8 Conclusion

Traditionally, programming has focused on performing complex operations sequentially on a single system and, as such, many of our existing programming languages are based on the assumption that programs must operate in this manner. In modern times, however, this paradigm is starting to fall apart as a rapidly increasing number of tasks involve the coordination of concurrent programs on various devices around the world. This leaves programmers with the complex task of trying to write and design such interconnected systems using tools that are unable to fully express such concepts—thus increasing how much work the programmer has to do while also making it possible for seemingly correct programs to err in unpredictable ways.

For several decades now, many theories have been proposed to address this issue by developing a system designed to reason about distributed and concurrent systems; however, many of these proposed calculi have some form of limit whether it be a mathematical limitation on what kinds of programs can be expressed (usually in order to attain certain correctness guarantees) or linguistically by utilizing a terse representation that is most accessible to those with a strong background in computer science theory. This provides two challenges: from the technical side, how can we develop an expressive language while maintaining correctness guarantees; and, from the humanities side, how can we make such a language open to a more general population of programmers?

This research was centered at the nexus of these two questions and, as such, the goal of this project was to develop a more expressive process calculus that programmers can use to communicate distributed, concurrent, and mobile tasks while retaining correctness guarantees. The result of this work can be viewed as providing two primary contributions: the development of a new framework for designing programming languages and Bismuth Calculus (a new process calculus creating using this framework and designed to address the project's goal).

New Language Design Framework The new programming language design framework this work introduces (Section 5) which provides a mechanism for the rapid low-cost evaluation and refinement of a programming language prototype based on audience. This is in contrast to many of the existing methods which either are high-cost and audience centered, or low-cost and designer centered. While this comes at the cost of the generalizability and granularity that these other methods have, in utilizing this framework throughout developing Bismuth Calculus, I was able to gain many valuable insights which enabled me to make various improvements to the language's design as well as note its limitations. As such, if deemed effective, this framework may serve as a valuable tool in helping diversify programming languages as coding goes from being a specialized skill to a general part of literacy [41].

Bismuth Calculus Based on its evaluation (Section 7.2), Bismuth Calculus is a fairly expressive calculus capable of representing a variety of tasks that a language in such a domain would need to be able to perform. Notably, Bismuth Calculus is able to provide a system for describing compositional systems where processes can contain (and interact with) multiple processes at once. To this extent, multiparty communication may be possible without a global state through the ability to join and merge channels through the introduction of helper processes which serve to transform channel protocols. In addition, as Bismuth Calculus is able to achieve this through the use of protocols which share a strong resemblance to regular language (Appendix E) for which programmers are already familiar with, this may make it easier for them to write protocols in the language, and it

may also make it possible for protocols to be automatically generated for them based on flowcharts which specific how systems should interact.

8.1 Limitations & Future Work

Despite these advances, several limitations exist with this project which would be worth investigating as part of future work:

1. **Verifying Framework Efficacy** - While this project appears to have benefited significantly from the use of the framework proposed in Section 5, without further studies, it is impossible to know for certain if the framework was able to accurately capture audience sentiment and to derive improvements from or if the framework is still too heavily biased by the researchers. To address this, I would recommend that a future work validates the results of this framework by either evaluating it in parallel with another well tested framework or by validating the framework's results through user studies.

2. **Parallel Operator within Protocols** - Currently, channels can only serve one protocol at a time and, while it is possible to treat channels as higher-order to enable one process to have multiple ways of communicating, this still results in a tree-like structure because each process can only have one primary channel. Introducing such a construct would also require us to find ways of splitting sequential channels into parallel operations (or developing a mechanism to implement a parallel loop protocol) so that concepts such as a multithreaded server can become possible.

3. **Abortable Channels** - In the evaluation, one of the language's major limitations was that of being unable to represent a protocol that allows for further communication to be preemptively closed by either process involved in the communication. Such a protocol would also have to ensure that any channels (or parallel subchannels) sent over an abortable channel are also abortable and will be automatically closed by the compiler if the primary channel is closed.

4. **Improved Notation** - As described in Section 7.2.2.4, the language's syntax and mechanisms for interacting with channels is inconsistent and may be challenging to work with. As these concepts are novel to this language and are fundamental to writing programs in it, finding better and more consistent metaphors and representations for the various concepts in the language would make it easier to work with and learn. In addition, especially in processes that interact with multiple complicated channels, it could be beneficial to find ways of making the language's notation better convey the state of the various channels as they change over time to make code easier to understand. Along these lines, I think it could also be interesting to explore other representations and tools for writing code in this language (such as developing a system for converting flow charts to protocols—thus allowing developers to visualize how their programs interact in a manner similar to a network map).

5. **Correctness Optimizations & Shared State Considerations** - As it stands, many potentially common operations in the language (such as protocol transforms) are incredibly inefficient due to their reliance on having intermediary processes that describe how to map between two protocols. While, in some cases, this may be necessary, it would be worthwhile to investigate if the language could be designed in a manner to make these kinds of operations easier and more efficient.

Similarly, it is apparent that there are many cases wherein processes may need to share access to a common resource (such as how multiple places in a codebase may wish to print text to the standard output stream). In the current language, such concepts are challenging to represent due to the lack of a global/multiparty state requiring us to manually split resources through the introduction of new processes. While many complications would exist with implementing a global state, it would be worthwhile to investigate ways of making multiparty resource sharing easier.

References

[1] Luke Church Alan F. Blackwell and Thomas Green. "The Abstract is 'an Enemy':Alternative Perspectives to Computational Thinking". In: (2008). URL: https://www.ppig.org/files/2008-PPIG-20th-blackwell.pdf.

[2] *API Routes: An Introduction — Next.js*. https://nextjs.org/docs/api-routes/introduction.

[3] Nick Benton, Luca Cardelli, and Cédric Fournet. "Modern Concurrency Abstractions for C#". In: *European Conference on Object-Oriented Programming*. 2002, pp. 415–440. URL: https://www.researchgate.net/profile/Nick-Benton/publication/220404627_Modern_Concurrency_Abstractions_for_C/links/0c9605264f818c9df8000000/Modern-Concurrency-Abstractions-for-C.pdf.

[4] David Bernstein. *Mobile Agents: An Introduction with Examples in Java*. https://w3.cs.jmu.edu/bernstdh/web/common/lectures/summary_mobile-agents.php.

[5] Ian Bogost. *Persuasive Games: The Expressive Power of Videogames*. The MIT Press, June 2007. ISBN: 9780262268912. DOI: 10.7551/mitpress/5334.001.0001. URL: https://doi.org/10.7551/mitpress/5334.001.0001.

[6] Rose Bohrer. "Lecture #7: Operational Semantics". 2022.

[7] Kevin Brock. *Rhetorical code studies: Discovering arguments in and around code*. University of Michigan Press, 2019. DOI: https://doi.org/10.3998/mpub.10019291. URL: http://library.oapen.org/handle/20.500.12657/23989.

[8] *Building a Single-Threaded Web Server - The Rust Programming Language*. https://doc.rust-lang.org/book/ch20-01-single-threaded.html.

[9] Luís Caires and Frank Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. Vol. 6269. Lecture Notes in Computer Science. Springer, 2010, pp. 222–236. DOI: 10.1007/978-3-642-15375-4_16. URL: https://doi.org/10.1007/978-3-642-15375-4_16.

[10] Luca Cardelli and Andrew D. Gordon. "Mobile Ambients". In: *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Maurice Nivat. Vol. 1378. Lecture Notes in Computer Science. Springer, 1998, pp. 140–155. DOI: 10.1007/BFb0053547. URL: https://doi.org/10.1007/BFb0053547.

[11] Michael Coblenz et al. "Interdisciplinary Programming Language Design". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 133–146. ISBN: 9781450360319. DOI: 10.1145/3276954.3276965. URL: https://doi.org/10.1145/3276954.3276965.

[12] Michael Coblenz et al. "PLIERS: A Process That Integrates User-Centered Methods into Programming Language Design". In: *ACM Trans. Comput.-Hum. Interact.* 28.4 (July 2021). ISSN: 1073-0516. DOI: 10.1145/3452379. URL: https://doi.org/10.1145/3452379.

[13] *Distributed Computing · The Julia Language*. https://docs.julialang.org/en/v1/stdlib/Distributed/. 2022.

[14] Cédric Fournet et al. "A Calculus of Mobile Agents". In: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*. Ed. by Ugo Montanari and Vladimiro Sassone. Vol. 1119. Lecture Notes in Computer Science. Springer, 1996, pp. 406–421. DOI: 10.1007/3-540-61604-7_67. URL: https://doi.org/10.1007/3-540-61604-7_67.

[15] Simon J. Gay and Vasco Thudichum Vasconcelos. "Linear type theory for asynchronous session types". In: *J. Funct. Program.* 20.1 (2010), pp. 19–50. DOI: 10.1017/S0956796809990268. URL: https://doi.org/10.1017/S0956796809990268.

[16] *Go RPC implementation using Go's standard http and rpc packages*. https://github.com/karankumarshreds/GoRPC. 2021.

[17] T.R.G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X. DOI: https://doi.org/10.1006/jvlc.1996.0009. URL: https://www.sciencedirect.com/science/article/pii/S1045926X96900099.

[18] Robert Harper. *Practical foundations for programming languages*. 2016. URL: https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf.

[19] Andrew Gary Darwin Holmes. "Researcher Positionality–A Consideration of Its Influence and Place in Qualitative Research–A New Researcher Guide." In: *Shanlax International Journal of Education* 8.4 (2020), pp. 1–10.

[20] *ipfs-pubsub-room - npm*. https://www.npmjs.com/package/ipfs-pubsub-room.

[21] Harvendra Kumar and A. K. Verma. "Comparative Study of Distributed Computing Paradigms". In: *BVICAM's International Journal of Information Technology* 1 (Jan. 2009). URL: https://www.researchgate.net/profile/Anil-Verma/publication/266176228_Comparative_Study_of_Distributed_Computing_Paradigms/links/55a3751008aeaefdb97bb962/Comparative-Study-of-Distributed-Computing-Paradigms.pdf.

[22] George Lakoff. "The contemporary theory of metaphor". In: (1993).

[23] George Lakoff and Mark Johnson. *Metaphors We Live By*. 1980.

[24] Yanhua Li. *Reliable Data Transmission*. https://users.wpi.edu/~yli15/courses/CS3516Fall20A/slides/CS3516-12-rdt1-3.pdf. 2020.

[25] Sam Lindley and J. Garrett Morris. "A Semantics for Propositions as Sessions". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 560–584. DOI: 10.1007/978-3-662-46669-8_23. URL: https://doi.org/10.1007/978-3-662-46669-8_23.

[26] Mei-Ling Liu. *Distributed Computing Paradigms*. 2007. URL: https://www.infor.uva.es/~jjalvarez/asignaturas/SD/lectures/chapter3.pdf.

[27] Michael Losonsky. *Linguistic turns in modern philosophy*. Cambridge University Press, 2006.

[28] Karl Mazurak and Steve Zdancewic. "Lolliproc: To Concurrency from Classical Linear Logic via Curry-Howard and Control". In: 45.9 (2010). ISSN: 0362-1340. DOI: 10.1145/1932681.1863551. URL: https://doi.org/10.1145/1932681.1863551.

[29] Robin Milner. *Functions as Processes*. Tech. rep. University of Edinburgh, 1990. DOI: https://link.springer.com/content/pdf/10.1007/bfb0032030.pdf.

[30] Michael Muller. "Curiosity, creativity, and surprise as analytic tools: Grounded theory method". In: *Ways of Knowing in HCI*. Springer, 2014, pp. 25–48.

[31] Stefan K. Muller and Hannah Ringler. "A Rhetorical Framework for Programming Language Evaluation". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 187–194. ISBN: 9781450381789. DOI: 10.1145/3426428.3426927. URL: https://doi.org/10.1145/3426428.3426927.

[32] Janet Horowitz Murray. *Hamlet on the holodeck : the future of narrative in cyberspace*. eng. New York: Free Press, 1997. ISBN: 0684827239.

[33] *Nuxt 3 - Server Engine*. https://v3.nuxtjs.org/guide/concepts/server-engine. 2022.

[34] John F. Pane. "Human-Centered Design of a Programming System for Children". In: (2001).

[35] Nancy Pennington. "Stimulus structures and mental representations in expert comprehension of computer programs". In: (1986). URL: https://apps.dtic.mil/sti/pdfs/ADA179392.pdf.

[36] *rpc package = net/rpc - Go Packages*. https://pkg.go.dev/net/rpc. 2023.

[37] David Sangiorgi David; Walker. *The pi-calculus : A Theory of Mobile Processes*. Cambridge University Press, 2003.

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

[39] Statista. *Number of hyperscale data centers worldwide from 2015 to 2021*. https://www.statista.com/statistics/633826/worldwide-hyperscale-data-center-numbers/. 2021.

[40] *TcpStream in std::net - Rust*. https://doc.rust-lang.org/std/net/struct.TcpStream.html.

[41] Annette Vee. *Coding Literacy: How Computer Programming is Changing Writing*. The MIT Press, 2017. DOI: https://doi.org/10.7551/mitpress/10655.001.0001.

[42] José M. Vidal. *Distributed Computing Paradigms*. URL: https://jmvidal.cse.sc.edu/talks/distobjects/distributedcomputingparadigms.html.

[43] Malte Viering et al. "A multiparty session typing discipline for fault-tolerant event-driven distributed programming". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–30. URL: https://dl.acm.org/doi/pdf/10.1145/3485501.

[44] Philip Wadler. "Is There a Use for Linear Logic?" In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*. Ed. by Charles Consel and Olivier Danvy. ACM, 1991, pp. 255–273. DOI: 10.1145/115865.115894. URL: https://doi.org/10.1145/115865.115894.

[45] Edgar Zamora-Gómez, Pedro García López, and Rubén Mondéjar. "Continuation Complexity: A Callback Hell for Distributed Systems". In: *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*. Ed. by Sascha Hunold et al. Vol. 9523. Lecture Notes in Computer Science. Springer, 2015, pp. 286–298. DOI: 10.1007/978-3-319-27308-2_24. URL: https://doi.org/10.1007/978-3-319-27308-2_24.

8.2 LaTeX Code Samples

- Acknowledgements environment from `https://latex.org/forum/viewtopic.php?t=5464`
- LLVM IR listings style from `https://github.com/mewmew/lstlangs/blob/master/llvm/lang.sty`
- JavaScript listings style from `https://github.com/ghammock/LaTeX_Listings_JavaScript_ES6`
- Rust listings style from `https://github.com/denki/listings-rust/blob/master/listings-rust.sty`
- Table 12 format from `https://tex.stackexchange.com/questions/377638/how-to-create-a-compact-comparison-table-like-this`

Glossary

C-style language Languages whose design has been inspired from or derived from the C programming language. Examples include C++, Java, and C#. 11

deadlock Deadlock is when two or more programs prevent each other from executing. This typically occurs when each program has exclusive access over at least one resource that both programs need in order to proceed—thus preventing either from making progress. 3, 6

DILL Dual intuitionistic linear logic - A form of logic where there are two contexts: one for linear resources and one for non-linear resources. 32

multithreading Multithreading is a form of shared-memory concurrency where the CPU is instructed to schedule multiple parts of a program to run at same time—giving our one program multiple "threads of execution". 3

mutual exclusion The principle in computer science where only one thread at a time is allowed access to a region of memory. 3

natural programming A formative design method for programming languages wherein programmers are asked to write code to solve a problem while being as creative as possible and using whatever language features they would like. Then, by imposing constraints on the language, researchers aim to uncover how programmers may naturally expect a language in such a domain to function—thus providing a basis for their own language design [12]. 16, 18

protocol fidelity A property of session typed languages which guarantees that programs will correctly and completely utilize their channels in manners consistent with the channel's protocol: a process must either completely recognize all channels that it interacts with or pass them off to another process which must do the same. This property arises as a consequence of these languages upholding preservation. . 36, 38, 39, 42

race conditions A race condition is when two processes modify the same resource at the same time—hence causing an unpredictable and undesirable result. 3, 6, 41

serialization Serialization describes the process of converting data in a program into some other form which can be stored (or sent to another program) and later parsed back into the same data. 2, 38

spinlock An inefficient technique for making a program wait for a certain condition to hold by repeatedly checking the condition until it becomes true due to work being accomplished in parallel. 39, 106

syntactic sugar An alternate representation of a language where each construct in the new representation's syntax is defined a combination of one or more constructs in the original language. This is typically done to make a language that is easier to work with while preserving the functionality of the original language. 44

Glossary of Symbols

! Of Course - Represents a resource for which another program controls how many times the resource is used.. 37, 42, 83

$\oplus$ Internal Choice - Used to represent a branching protocol for which the local process decides which branch to take. 44

$\otimes$ Multiplicative Conjunction/Tensor - Used to join multiple linear resources together as one resource. $\neg(A \otimes B) \equiv \neg A \parr \neg B$. 35

$\parr$ Multiplicative Disjunction/Par - Used to represent having multiple linear resources at once which must be split apart in order to be used. $\neg(A \parr B) \equiv \neg A \otimes \neg B$. 35

& With/External Choice - Used to represent a branching protocol for which another process decides which branch to take. 44

? Why Not - Represents a resource for which the local program controls how many times the resource is used.. 42, 83

A Table of the Cognitive Dimensions Discussion Tools

Table 4: Summary of the Cognitive Dimensions Framework [17]

Dimension Name	Description
Abstraction Gradient	How well the language balances abstractions to prevent the user from needing to implement or interact with too many, while also not having so few as to prevent users from creating abstractions.
Closeness of Mapping	How directly tasks in the language's problem domain can be mapped to code in the language.
Consistency	How well someone who has limited knowledge of the language can infer the meaning of other structures in the language.
Diffuseness/Terseness	How compactly can a programming language describe a task while also ensuring that the language is no so terse that it is challenging to scan as programs all appear similar?
Error-proneness	How prone the language is to mistakes (such as typos and mismatched scopes) where the programmer knew what they wanted to do and how to accomplish it in the language, but a simple mistake caused an error in the program.
Hard Mental Operations	At the notational level of the language, is there a way of grouping language constructs in such a way that they become incomprehensible when the concept itself is not hard to understand?
Hidden Dependencies	Are there unclear dependencies between language constructs (such as side-effects)?
Premature Commitment	Is the programmer forced to make decisions prior to having adequate information to do so?
Progressive Evaluation	Can the programmer evaluate partially completed programs to determine if they are on the right track or not?
Role-expressiveness	Is it possible to look at a small snippet of code and understand, generally, what it is for?
Secondary Notation	Does the language support things like commenting, indenting, and naming conventions which allow the programmer to impart additional meaning to the program which may help others read and understand it?
Viscosity	How much work does the programmer have to put in to make a small change?
Visibility and Juxtaposability	How readily can the programmer find, access, and compare various sections of code?

B Case Study

B.1 Language Benefits & Limitations

B.1.1 π-Calculus

Table 5: π-Calculus Benefits and Limitations

Benefits	Limitations
• Compositional • Can model functions • Can model servers	• Untyped • Lacks data types • Global channels • Static environment: repeated processes repeat forever

B.1.2 Ambient Calculus

Table 6: Ambient Calculus Benefits and Limitations

Benefits	Limitations
• Compositional • Can model functions • Can model servers • Higher-order processes • Describes the location of resources • Local communication • Assigns capabilities to ambients for security	• Untyped • Lacks data types • Static environment: repeated processes repeat forever • Unrealistic properties such as ambient name collision

B.1.3 Lolliproc

Table 7: Lolliproc Benefits and Limitations

Benefits	Limitations
• Typed (Including communication channels) • Can model functions • Higher-order processes • Describes the location of resources • Deadlock-free	• Non-compositional • Hard to represent servers • Global channels

B.1.4 πDILL

Table 8: πDILL Benefits and Limitations

Benefits	Limitations
• Typed • Allows for both persistent and non-persistent linear resources • Deadlock free • Describes location of resources • Linear & non-linear resources • Can model functions	• Not compositional • Uses global channels • Harder to model servers

B.1.5 GV

Table 9: GV Benefits and Limitations

Benefits	Limitations
• Typed • Allows for both persistent and non-persistent linear resources • Deadlock free • Can model functions • Describes location of resources • Describes resources linearly	• Harder to model servers/long term interactions • Does not support non-linear local resources

B.1.6 Functional GV

Table 10: Functional GV Benefits and Limitations

Benefits	Limitations
• Typed • Allows for both persistent and non-persistent linear resources • Deadlock free • Can model functions • Describes location of resources	• Deals primarily with threads and distinguishes the capabilities of the main thread from children • Harder to model servers/long term interactions

B.2 Summary Feature Set

Table 11: Process Calculus Language Features

Feature	Description	Present in	Absent from/Challenging In
Compositional	Programs in the calculus can be composed in any manner desired. For instance, they can act as the client to any number of processes and server for any number of processes.	• π-calculus • Ambient calculus	• Lolliproc • πDILL
Typed	Allows us to assign types to resources and channels to describe what they are, how they function, and how we can interact with them.	• Lolliproc • πDILL • GV • Functional GV	• π-calculus • Ambient calculus
Model Functions	Programs in the calculus can act as function: given an input, they can perform short-term operations (such as sorting) before resulting in a value and terminating	• π-calculus • Ambient calculus • Lolliproc • πDILL • GV • Functional GV	
Model Servers	Programs in the calculus can act on a long-term scale: interacting with an arbitrary number of other processes by sending, receiving, and performing operations over an arbitrarily long duration of time.	• π-calculus • Ambient calculus	• Lolliproc • πDILL • GV • Functional GV

Continued on next page...

Process Calculus Language Features Continued

Feature	Description	Present in	Absent from/Challenging In
Local Channels	Language represents channel in a manner such that only the two processes communicating across a channel have access to it. Without this (when we have global channels/mechanisms for communication), it can become incredibly easy to break the language's communication system through misusing it.	• Ambient calculus	• π-calculus • Lolliproc • πDILL
Repeated Process Termination	While repetition is critical to computing, in the real work, processes are likely to eventually need to terminate and, as such, a process calculus should be able to describe how all processes would terminate (regardless of if they ever do).	• Lolliproc	• π-calculus • Ambient Calculus
Higher-order Processes	Describes the ability to send a process/program to another process (analogous to higher-order functions). Languages who are able to achieve similar results due to channel-passing are not included.	• Ambient Calculus • Lolliproc	• π-calculus

Continued on next page...

Process Calculus Language Features Continued

Feature	Description	Present in	Absent from/Challenging In
Describes Resource Location	The calculus allows us to describe the location of the program's resource. Being able to do this significantly lowers ambiguity given the higher-order nature of channels.	• Ambient Calculus • Lolliproc • πDILL • GV • Functional GV	• π-calculus
Deadlock Free	Deadlock can never occur in any of the programs written in the language.	• Lolliproc • πDILL • GV • Functional GV	• π-calculus • Ambient Calculus

B.3 Case Study Results Summary

Table 12: Case Study Summary

Language	Communications			Interactions			Resources		
	Does not use global state Link exactly two processes	Typed	Easily models complex protocols Describe location of resources Occurs consistently (regardless of type)	Short-term interactions Long-term interactions Described consistently	Support repetition	Not persistent	Deadlock free Compositional Fault tolerant	Higher-order programs (mobility) Higher-order channels Typed	Non-linear local resources Channels cannot be duplicated Channels cannot be discarded
π-calculus	○	○	○	●	○	●	○	●	○	○	◐	○	○
Ambient calculus	●	●	◐	●	●	●	●	○	○	●	○	◐	●	●	◐	◐	○
Lolliproc	◐	○	●	○	●	◐	●	◐	●	●	○	○	●	●	●	◐	◐	●
πDILL	●	○	●	○	●	●	●	◐	●	◐	●	●	●	○	○	●	●	●	●
GV	●	○	●	○	●	●	●	◐	●	◐	●	●	●	○	○	●	●	●	○	●	●
Functional GV	●	○	●	○	●	●	●	◐	◐	◐	●	●	●	○	○	●	●	◐	○	●	●
Distributed Join-Calculus	●	●	○	-	●	-	●	◐	●	-	-	-	-	-	●	◐	●	●	◐	-	○
Bismuth Calculus	●	●	●	◐	●	◐	●	●	●	●	●	◐	●	○	●	●	●	●	●	●	●

● = Provides property ◐ = Indirectly provides property/Challenging to achieve
○ = Does not provide property - = Not evaluated

C Baseline Calculus Definition

The following are the initial set of rules for my language. While the language defined by these rules is not sound, they were able to serve as a baseline for my corpus study as many of the rules could exist in a sound language, and they provide enough of an approximation for what kinds of programs would be possible in my language, and thus allowing me to get a better idea of how my language may need to change prior to having completed a full soundness proof of it.

C.1 Send

$$\text{(TSend)} \quad \frac{\Gamma; \Delta \vdash e : \tau \quad \Gamma; \Delta\{e\}, c : \restriction \rho \restriction \vdash P}{\Gamma; \Delta, c : \restriction \neg \tau; \rho \restriction \vdash c\langle e \rangle.P}$$

$$\text{(OPSend)} \quad (G; L_i, c; c\langle e \rangle.P) \mapsto (G'; L_i\{e\}, c; P) \qquad \begin{pmatrix} (G; L_i) \vdash e \Downarrow v \\ G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c} \mapsto \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c} \mapsto \overrightarrow{m}, v \end{pmatrix}$$

C.2 Receive

$$\text{(TRecv)} \quad \frac{\Gamma\langle x \rangle; \Delta\langle x \rangle, c : \restriction \rho \restriction \vdash P}{\Gamma; \Delta, c : \restriction + \tau; \rho \restriction \vdash c(x).P}$$

$$\text{(OPRecv)} \quad (G; L_i, c; c(x).P) \mapsto (G'; L_i[x \mapsto v], c; P) \qquad \begin{pmatrix} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto v, \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{pmatrix}$$

C.3 External Choice (Offer)

$$\text{(TCase)} \quad \frac{\Gamma; \Delta, c : \restriction \rho_1 \restriction \vdash P.R \quad \Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash Q.R}{\Gamma; \Delta, c : \restriction \rho_1 \,\&\, \rho_2 \restriction \vdash c.\mathtt{case}(P, Q).R}$$

$$\text{(OPCase1)} \quad (G; L_i, c; c.\mathtt{case}(P, Q).R) \mapsto (G'; L_i, c; P.R) \quad {}^{1}$$

$$\text{(OPCase2)} \quad (G; L_i, c; c.\mathtt{case}(P, Q).R) \mapsto (G'; L_i, c; Q.R) \quad {}^{2}$$

$$^{1} \begin{pmatrix} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{SEL}[1], \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{pmatrix}$$

$$^{2} \begin{pmatrix} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{SEL}[2], \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{pmatrix}$$

C.4 Internal Choice (Project/Select)

$$\text{(TSelect1)} \quad \frac{\Gamma; \Delta, c : \restriction \rho_1 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \rho_1 \oplus \rho_2 \restriction \vdash c[1].P}$$

$$\text{(OPSelect1)} \quad (G; L_i, c; c[1].P) \mapsto (G'; L_i, c; P) \qquad \begin{pmatrix} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c} \mapsto \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[1] \end{pmatrix}$$

(TSelect2) $\dfrac{\Gamma; \Delta, c : \restriction \rho_2 \upharpoonright \vdash P}{\Gamma; \Delta, c : \restriction \rho_1 \oplus \rho_2 \upharpoonright \vdash c[2].P}$

(OPSelect2) $(G; L_i, c; c[2].P) \mapsto (G'; L_i, c; P) \quad \begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m} \\ G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, \mathsf{SEL}[2] \end{pmatrix}$

C.5 While Loop

(TWhile) $\dfrac{\Gamma; \cdot \vdash e : \mathsf{bool} \quad \Gamma; \Delta_1 \vdash P \quad \Gamma; \Delta \vdash Q}{\Gamma; \Delta \vdash \texttt{while}(e)\{P\}.Q}$ [1]

(OPWhileT) $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; P.\texttt{while}(e)\{P\}.Q)$ [2]

(OPWhileF) $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; Q)$ [3]

---

[1] $(\Delta_1 \equiv \{x_i : \restriction ?\rho_i \upharpoonright | \Delta(x_i) = \restriction ?\rho_i; \rho_i' \upharpoonright \text{ for some } \rho_i'\})$
[2] $((G; L_i) \vdash e \Downarrow \mathsf{true})$
[3] $((G; L_i) \vdash e \Downarrow \mathsf{false})$

C.6 Unfold

(TUnfold) $\dfrac{\Gamma; \Delta, c : \restriction \rho_1; ?\rho_1; \rho_2 \upharpoonright \vdash P}{\Gamma; \Delta, c : \restriction ?\rho_1; \rho_2 \upharpoonright \vdash \texttt{more}(c).P}$

(OPUnfold) $(G; L_i; \texttt{more}(c).P) \mapsto (G'; L_i; P)$ [1]

---

[1] $\left(G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m} G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, \mathsf{START{-}LOOP}\right)$

C.7 Weaken

(TWeaken) $\dfrac{\Gamma; \Delta, c : \restriction \rho_2 \upharpoonright \vdash P}{\Gamma; \Delta, c : \restriction ?\rho_1; \rho_2 \upharpoonright \vdash \texttt{weaken}(c).P}$

(OPWeaken) $(G; L_i, c; \texttt{weaken}(c).P) \mapsto (G'; L_i, c; P)$ [1]

---

[1] $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m} \\ G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, \mathsf{END{-}LOOP}[] \end{pmatrix}$

C.8 Accept

$$\text{(TAccept)} \quad \frac{\Gamma; \Delta_1, c : \uparrow \rho_1 \uparrow \vdash P \quad \Gamma; \Delta, c : \uparrow \rho_2 \uparrow \vdash Q}{\Gamma; \Delta, c : \uparrow ! \rho_1; \rho_2 \uparrow \vdash \texttt{accept}(c)\{P\}.Q} \quad {}^{1}$$

$$\text{(OPAccept-Loop)} \quad (G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{accept}(c)\{P\}.Q) \quad {}^{2}$$

$$\text{(OPAccept-End)} \quad (G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; Q) \quad {}^{3}$$

---

1 $(\Delta_1 \equiv \{x_i : \uparrow ? \rho_i \uparrow | \Delta(x_i) = \uparrow ?? \rho_i; \rho_i' \uparrow \text{ for some } \rho_i'\})$

2 $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{pmatrix}$

3 $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{END{-}LOOP}[], \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{pmatrix}$

C.9 Close Channel

$$\text{(TClose)} \quad \frac{\Gamma; \Delta \vdash P}{\Gamma; \Delta, c : \uparrow \uparrow \vdash P}$$

$$\text{(OPClose)} \quad (L_i, c; P) \mapsto (L_i; P)$$

C.10 Close Process

$$\text{(TExit)} \quad \frac{}{\Gamma; \cdot \vdash \texttt{exit}}$$

$$\text{(OPExit-Continue)} \quad (G; \overrightarrow{L}; (P_1, \ldots, \texttt{exit}, \ldots, P_n)) \mapsto (G; \overrightarrow{L}; (P_1, \ldots, P_n))$$

$$\text{(OPExit-Isolated)} \quad (G; \overrightarrow{L}; \texttt{exit})$$

C.11 Execute Program

$$\text{(TExec-Prog)}\quad \frac{\Gamma; \Delta, c : \restriction \neg \rho \restriction \vdash Q \quad \Gamma'; \Delta', n : \restriction \rho \restriction \vdash P}{\Gamma; \Delta \vdash (c = \texttt{exec}\ \ P :: n : \rho).Q}$$

$$\text{(OPExec-Prog)}\quad (G; L_i; (c = \texttt{exec}\ \ P :: n : \rho).Q) \mapsto (G'; (L'_i, L'_P); (P, Q)) \quad {}^{1}$$

$$\text{(TExec-Var)}\quad \frac{\Gamma; \Delta \vdash P :: n : \rho \quad \Gamma; \Delta\{P\}, c : \restriction \neg \rho \restriction \vdash Q \quad \Gamma'; \Delta', n : \restriction \rho \restriction \vdash P}{\Gamma; \Delta \vdash (c = \texttt{exec}\ \ P).Q}$$

$$\text{(OPExec-Var)}\quad (G; L_i; (c = \texttt{exec}\ \ P).Q) \mapsto (G'; (L'_i, L_P); (P, Q)) \quad {}^{2}$$

$$1 \quad \begin{pmatrix} G \equiv \overrightarrow{z \mapsto q} \\ G' \equiv \overrightarrow{z \mapsto q}, fn_c \mapsto [], fn_n \mapsto [] \\ L'_i \equiv L_i[c \mapsto fn_c] \\ L'_P \equiv \{n \mapsto fn_n\} \\ \overline{c} = n \\ \overline{n} = c \end{pmatrix}$$

$$2 \quad \begin{pmatrix} G \equiv \overrightarrow{z \mapsto q} \\ G' \equiv \overrightarrow{z \mapsto q}, fn_c \mapsto [], fn_n \mapsto [] \\ L'_i \equiv L_i[c \mapsto fn_c] \\ \overline{c} = n \\ \overline{n} = c \end{pmatrix}$$

C.12 Link (Cut)

$$\text{(TLink)}\quad \frac{\Gamma; \Delta, a : \uparrow\rho_2\uparrow, b : \uparrow\rho_3\uparrow \vdash P}{\Gamma; \Delta, a : \uparrow\rho_1; \rho_2\uparrow, b : \uparrow\neg\rho_1; \rho_3\uparrow \vdash \texttt{link}(a, b).P}$$

Case: $\rho_1 \equiv +\tau$
(OPLink-Recv) $(L_i; \texttt{link}(a, b).P) \mapsto (L_i; b\langle a()\rangle.P)$

Case: $\rho_1 \equiv -\tau$
(OPLink-Send) $(L_i; \texttt{link}(a, b).P) \mapsto (L_i; a\langle b()\rangle.P)$

Case: $\rho_1 \equiv ?\rho$
(OPLink-While) $(L_i; \texttt{link}(a, b).P) \mapsto (L_i; \texttt{accept}(b)\{\texttt{more}(a).\texttt{link}(a, b)\}.\texttt{weaken}(a).P)$

Case: $\rho_1 \equiv !\rho$
(OPLink-Accept) $(L_i; \texttt{link}(a, b).P) \mapsto (L_i; \texttt{accept}(a)\{\texttt{more}(b).\texttt{link}(a, b)\}.\texttt{weaken}(b).P)$

Case: $\rho_1 \equiv \rho_a \,\&\, \rho_b$
(OPLink-Ext) $(L_i; \texttt{link}(a, b).P) \mapsto (L_i; a.\texttt{case}(b[1].\texttt{link}(a, b), b[2].\texttt{link}(a, b)).P)$

Case: $\rho_1 \equiv \rho_a \oplus \rho_b$
(OPLink-Int) $(L_i; \texttt{link}(a, b).P) \mapsto (L_i; b.\texttt{case}(a[1].\texttt{link}(a, b), a[2].\texttt{link}(a, b)).P)$

D Corpus Study

D.1 General Notes from Writing Code in the Calculus

- Lack of names in protocols make them harder to understand than function parameters
- Protocols can become tiring to write and seemingly redundant
- Having to write out the protocols can make it harder to change what the program does
- The lack of global variables makes things like importing printf more complicated along with resource sharing
  - This means that it takes more code (and more computations) to process the same thing that other langs may find easy (ie globals, error handling/early disconnects)
  - It also, however, means that we have to document exactly how processes interact
  - Unfortunately, persistent resources are harder to understand the interactions of than non-persistent channels due to how their use can be more arbitrary and based on control flow
- It is possible to define logical and valid programs whose channel type is impossible to represent
  - This is because the language itself is more expressive than the language's types, and, as the type system cannot be turing complete while remaining type-safe, we don't have a good way to represent these types (at time of definition)
  - One workaround would be to introduce protocol inference as, once we use a process, even with a complex type, it should become describable
  - One downside to this, however, is that it means that the types of a program no longer can be used to easily document its actions
- Channels themselves can be a bit clunky to use as they require a more verbose syntax than functions
  - This somewhat makes sense due to how complex program interactions may have to be; however, for simple cases (like functions), it seems excessive
  - I somewhat wonder if naming each element in the protocol would help resolve this (as we may be able to automatically handle the interactions); however, I don't think this would be able to capture everything, and I'm not sure how well it'd work
- The bulky channels/processes (along with their pass-by-value like operations) would seem to suggest that a non-linear version of a program (a function) which uses pass-by-reference may be useful
- In some cases, programs may be able to be curried into functions to help make operations more efficient
- Still not exactly sure how channels would work with case, auto parse (might need some level of reflections), and serialization (also, potentially details about how linear resources would work and compare to non-linear)

- Some operations like link seem to be ambiguous on what they actually mean/how they actually function
- Syntax for case select is bad—seems like an array and I'm not really sure what's the best way to select a case
  - It can't exactly be by number (or naming each channel) (without syntactic sugar) due to the fact that programs may have cases written in different orders or have additional options that the other program is unaware of
  - Forcing one to write out the channel selected seems excessive due to the fact that channels can be long, and they can be written in multiple ways
- Channels (even with exit) perform differently than return does due to the fact that returns know that nothing else has been previously returned and channels might not know this (see example program)
- Need a way to fold programs (ie, convert things to be more functional) as otherwise type-checking is incredibly challenging due to the use of multiple channels (albeit, there may be a way to work around this; it would just be complicated)
- How clear are channel operations like exec? (Thinking of pennington study)
- Use of persistent channels (really splitting them) is a scheduling problem
- Inconsistent use of keywords
  - Exec - is this a function? if so, why doesn't it use function syntax?
  - Case/Select/More/Etc - These all use various syntaxes to the point where some are function calls on a channel, and others are passing the channel to a function. Why?
- Somewhat challenging (if not impossible (at least without abstractions)) to handle mobile-agent like cases where there is a disconnect and reconnect during computation to truly make them async. (Work can occur offline!)
- Is there a way for two clients to negotiate what state they should be in? Ie, both have to verify channel state independently? This would seem impossible as we only have int and ext choice (not simultaneous) so, one would have to take precedence over the other?
- How can parallel split really work?
  - First of all, much of the channel algebra is incredibly intensive when it could be handled by just the type system (as it requires intermediate processes to help make things work)
  - Second, for parallel split, it'd have to mutex one channel or somehow multiplex it. Not sure how we can get such complex behavior (esp while keeping things safe)
  - For mobile processes, if we send something (like the identifier) and not the code, how do we know that what gets executed is what we want? or is that one of the trade-offs we have to assume? How can we even be sure that the other program has the desired program? is that part of the type system (and handled as a disconnect due to protocol mismatch failure)?

- Just as the type for a recursive function can't be written, I don't think a recursive channel is possible
  - While we do have loops instead, something about this does seem off as recursive calls can be a slightly different format than loops—thus giving them different meanings—albeit, I'm not sure how possible such a type really is (outside of doing something like ferrite where a dedicated operator/type exists for them)
  - Along these lines, what does it say about my language, given there are no globals/closures, and thus the recursive channels aren't really possible?
  - Should, maybe, functions be closures except for linear resources? This seems inconsistent though. What does it say about the language given that persistent types/definitions can permeate into other definitions, but not any other types/data—especially when considering that this breaks the hierarchy of type properties?
  - Persistent channels != persistent linear resources due to channels being temporal
  - How do interlocking/interweaving channels work? some of this starts to become a scheduling problem
- How do linear types work with RDT?
  - This would seem to have issues given that linear types are linear and rdt may require retransmission
  - Albeit, this may be an extension of the case where, if a program errors, it has to return its linear types back to be handled by someone else (although, how would this be handled with things like division where its so common, and a pain to recover from?)
  - Maybe we have to treat linear types as non-linear keys that map to the channel somewhere and then some program (thats part of the language) is responsible for handling these things, but what does it say about the language that this kind of operation might not be possible? Similarly, we have to consider how this works with being able to parse data automatically while our language doesn't really have casting per say.... while, at the same time, some casts are implicit by typing with sums?
- Do we need to have process/channels, or can we just have buffers?
  - This would make the lang more similar to the existing works by disconnecting channels from processes and making them less of a pain (ie tied into everything), but I'm nit sure if this makes sense? Sure, we are typing the buffers, but, its certainty a complex buffer with interactions at various places, an undefined length, and the potential for multiplexing...
- How would we integrate with things like system hooks/functions which are frequently based on having a shared global state (or mutable variables)? Having a global state is just such a common concept that having to interact with one may be odd because we don't just have one channel...
- How would the main program work? It needs to be given stdin, stderr, stdout, and the ability to return a status code..... Plus, we can't just import io streams..... Files have a similar problem, but I think they're more easy to handwave away by saying that there exists

a program which gets a channel to a file... maybe? This still has some notion of global state though

- How would something like a thread pool work? Or I guess, can thread pools not pause/resume execution? Along these lines, should we allow for some notion of pause/resume execution? I don't know if this could be done automatically (linear resources probably make this hard), but requiring the programmer to specify how this works in all cases may be annoying (but also necessary)?

From Paradigms:

- How would something like a server be modeled? Who gets to control the loop?
- Can't really do exception handling when we have linear resources as we don't know when they will be consumed/we have to make sure they are consumed despite async commands. And, even stuff like division becomes a pain...
- May have to use pre/post conditions to make loop rules more flexible in terms of allowing for stuff like accessing ext vars (and potentially just using subexponentials)
- Regex can be misleading as multiple ways to diagram same exact regex. Also, there are times where seq seems like an or. Ie, ?P;P. This seems like loop and at each iteration, give me the chance to do something else, when its not exactly...?
- what if process discovery fails?
- Channel algebra is challenging and computationally intensive
- linear hashmap: have everything have a destructor process?
- It can sometimes be challenging to follow what type a protocol has as it changes throughout the code and is only explicitly stated during the channel's definition. Because of this, one must mentally track all operations that are applied to the channel to determine its current state.

D.2 Client-Server Model

Description

The client-server model is a common paradigm wherein one program, known as the server, listens for and responds to requests made by other programs, known as clients.

Code

Program 1: Rust Client-Server Implementation [8, 40]

```rust
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

// Server
fn main() {                                                    // 1
    let listener = TcpListener::bind(<address>).unwrap();      // 2

    for stream in listener.incoming() {                        // 3
        let mut stream = stream.unwrap();

        let buf_reader = BufReader::new(&mut stream);
        let http_request: Vec<_> = buf_reader
            .lines()
            .map(|result| result.unwrap())
            .take_while(|line| !line.is_empty())
            .collect();

        stream.write_all(...).unwrap();
    }
}

// Client
fn main () {                                                   // 1
    let mut stream = TcpStream::connect(<address>)?;           // 4
    stream.write(...)?;
    stream.read(...)?;                                         // 5
}
```

Program 2: Bismuth Calculus Client-Server Implementation

```
1  define Server :: c : ... = {
2      define ServerHandler :: io : Channel<!(+Message;-Response)> = {
3          accept(io) {
4              Message m = io.recv();
5              Response r = ....;
6              io.send(r);
7          }
8      }
9
10     Listen(<address>, exec ServerHandler);
11 }
12
13 define Client :: c : ... = {
14     Channel<-Message;+Response> io = connect ....;
15     Message m = ...;
16     io.send(m);
17     Response r = io.recv();
18 }
```

1 2 3 2 1 4 5

Comments

1. In both versions of the program, this represents the server or client process. Both languages treat the type of this function/process separately from the mechanics of the server/client: in the Rust implementation, the function's signature says nothing about the server/client code that it will be processing, and in my language, the process's signature says nothing about the server/client functionality either.

2. Here is where the programs open up a websocket that they will use to serve the client's requests. Notably, in the Rust example, this occurs before the request handling code. In contrast, in my language, this occurs after we have defined a new process for handling server requests. In my version, this allows for us to abstract away code that listens for and parses server messages as we know the protocol type of our channel. As a downside, however, we now must respect this protocol, and the server lacks a mechanism to close the connection until all clients have been served. This is because the server's protocol follows an ! instead of a ?. This is because, in the client-server model, the clients connect to and are responsible for sending the first message. As such, the server must respect the clients and process all of their requests (failure to do so would violate linearity principles). Not only does this inability for the server to abort a connection serve as a limitation, it also makes writing these processes a bit more complicated as one may initially think that the server should be responsible for choosing how long it is able to run for. In addition, cases like these highlight the fact that, to implement a server in my language, we need a separate process for handling the server code (while, in languages like Rust, this can all be done within one function). This also provides us with the added complexity that, when executing a subprocess, we are given a channel that is inverse to the program's protocol. As such, when we pass a channel to our handler program to the Listen operation (which, we are assuming is a built-in function providing reliable data transmission), the channel signature in Listen is the inverse of the protocol for the program that we need to define. As such, looking at the types required by an operation to determine what one has to implement can be misleading.

3. In both programs, this block of code is responsible for reading and replying to each of the requests made to our server. The main differences here are:
    - In the Rust version, data parsing has to be done automatically (where as, my language, theoretically, could automatically handle this as the data sent over the channel is governed by a protocol).
    - In the Rust version, there is the ability to handle errors and for the server to abort the connection whereas, in my language, this is currently impossible.
4. Here is where the client connects to the server. In my language, this commits the client to follow the entire protocol (and thus, potentially enables automatic parsing of data). Whereas, in Rust, clients are not obligated to follow a protocol (thus enabling error handling) and are resposible for processing the data manually.
5. This is where the clients interact with the server via reading/writing from the channel. The main difference here appears to be how my language would allow for users to directly send and receive data based on the protocol's type where, in Rust, these conversions would have to be written by the developer.

D.3 Publish/Subscribe Model

Description

In a Publish/Subscribe model, processes communicate with each other by broadcasting messages which are labeled as having a specific room. Then, by subscribing to specific rooms, processes are able to receive these messages.

Code

Program 3: JavaScript Peer-to-Peer Publish/Subscribe Client using IPFS/Libp2p [20]

```
const Room = require('ipfs-pubsub-room')
const Libp2p = require('libp2p')

const libp2p = new Libp2p(...) // Handles interaction with peers via routing server
await libp2p.start()

const room = new Room(libp2p, <topic name>) // Subscribe to a topic
room.on('message', (message) => {...}) // Receive messages on topic
room.broadcast(...) // Broadcast message to peers
```

Program 4: Bismuth Calculus Publish/Subscribe

```
define PubsubServer :: c : Channel<
                            !(+String;
                              +(broadcast: Channel<!+Message>
                              | receive: Channel<?-Message>)
                            )> = {

    LinearMap<String, ?(-(Channel<!+Message> | Channel<?-Message>))> map = ...

    accept(c) {
        String topic = c.recv();
        var env = c.recv();

        if map.has(topic) as helper {
            more(helper);
            helper.send(env);
        } else {
            Channel<?(-(broadcast: Channel<!+Message> | receive: Channel<?-Message>)
    )> helper = exec PubSubHelper;
            map.put(topic, helper);
            more(helper);
            helper.send(env);
        }
    }
}
```

1 2 3

Program 5: Bismuth Calculus Publish/Subscribe - Helper Process

```
1  define struct Message { ... }
2
3  define PubSubHelper :: c : Channel<
4                              +Channel<!(
5                                  +(broadcast: Channel<!+Message>
6                                  | receive: Channel<?-Message>))>
7                             > = {
8      var rest = c.recv();
9
10     ((Channel<!+Message> | Channel<?-Message>) + Unit) envOpt = Unit;
11
12     accept(rest, 1) {
13         envOpt = rest.recv();
14     }
15
16     match envOpt
17         | Unit u => {
18                 accept(rest) {
19                     (b: Channel<!+Message> | r: Channel<?-Message>) = rest.recv();
20                     link(b, r);
21                 }
22                 exit;
23           }
24         | (broadcast: Channel<!+Message> | receive: Channel<?-Message>) => {
25                 parallel:
26                     |   (rest, broadcast, receive) => {
27                             accept(rest) {
28                                 (b: Channel<!+Message> | r: Channel<?-Message>) =
       rest.recv();
29
30                                 broadcast = merge(broadcast, b);
31                                 receive = merge(receive, r);
32                             }
33                         }
34                     |   (broadcast, receive) => {
35                             accept(broadcast) {
36                                 receive.send(broadcast.recv())
37                             }
38                         }
39                 weaken(receive);
40                 exit;
41             }
42 }
```

1 4 5 6 7

Comments

1. Channels like this can be hard to read
2. While the language does not have linear maps, there is nothing to suggest that such a concept would be impossible. The real representation of such a concept would likely be much more complicated than this to ensure all linear resources are used; however, without an expanded language, using a basic syntax like this serves as an approximation.

3. One downside of this approach is that we are using one thread per pubsub room. It may be possible for there to be a workaround to this; however, it would likely be more complicated to mix threads and sequential execution.

4. It is inconvenient having to use optionals here, but we have no choice

5. This concept does not currently exist in our language; I am using it as a shorthand for an acceptWhile that only accepts one element.

6. Code in this branch is mostly dead as the only way to get here is if our accept loop closes without sending us any data (which, from the definition of the server, we know is impossible). This, however, is impossible to prove in general and thus, we have to have this case to go through the motions of using our linear resources.

7. This concept does not exist in the current version of the language, but it is needed in order to make this system work. The basic idea is that we can distribute our linear resources among a variety of processes (? and ! channels can potentially be used at the same time via a mutex). It also becomes unclear what would happen if broadcast were to close early (and if that would case a problem). Similarly, we have to determine under what circumstances a protocol of type ! can be split across a parallel operator and how that would be managed. Finally, parallel operators like this could potentially open the door for deadlocks (if they aren't already possible).

D.4 Remote Procedure Call (RPC)

Description

Remote Procedure Calls describe the paradigm wherein functions can be dispatched on remote systems instead of the local system.

Code

Program 6: Go RPC (using Library) [16, 36]

```
1  import (
2      "net"
3      "net/http"
4      "net/rpc"
5  )
6  type Args struct { ... }                                          1
7
8  // Client
9  func main () {
10     var reply <reply type>                                        1
11     args := Args{...}
12
13     client, err := rpc.dial(<proto>, <address>)
14     if err != nil {
15         // Handle error                  2                 1
16     } else {
17         err := client.Call(<function name>, args, &reply)
18
19         if err != nil {
20             // Handle Error
21         }
22     }
23 }
24
25 // Server
26 type ServerResponse <response type>                               1
27
28 func (t *ServerResponse) <function name>(args *Args, reply *<reply type>) error {
29     *reply = ...                                                                  2
30     return nil
31 }
32
33 func main () {
34     server := new(ServerResponse)
35     rpc.Register(server)
36
37     rpc.HandleHTTP()
38
39     listener, err := net.Listen(<proto>, <address>)               3
40     if err != nil {
41         // Handle Error
42     }
43
44     http.Serve(listener, err)
45 }
```

Program 7: Bismuth Calculus RPC

```
1 define struct FnArgs1 = { int a; int b; }
2 define fn Fn1 ( int a, int b ) : int { ... }
3
4 define struct FnArgs2 = { string s }
5 define fn Fn2  (string s) : boolean { ... }
6
7 define RPCServer :: c : Channel<!(ExternalChoice<+FnArgs1;-int, +FnArgs2;-boolean>)>
      = {
8     accept(c) {
9         c.case(
10            {
11                FnArgs1 args = c.recv();
12                c.send(Fn1(args.a, args.b));
13            },
14            {
15                FnArgs2 args = c.recv();
16                c.send(Fn2(args.s));
17            }
18        )
19    }
20 }
21
22 define RPCClient :: c : Channel<InternalChoice<-FnArgs1;+int, -FnArgs2;+boolean>) =
      {
23    c[1];
24    c.send(FnArgs1(1, 2));
25    int ans = c.recv();
26 }
```

2

1

3

2

Comments

1. One key difference between RPC in Go and my language is that, in Go, RPC is set up in a manner where the parameters of the procedure and the resulting type are separate whereas, in my language, the protocol ensures that the two are used together. In addition, the use of a protocol in my language means that we can define a variety of possible functions that can be called and treat them all as one protocol, whereas, in Go, it appears that we may have to treat each possible RPC call separately.

2. In Go, the library for handling RPC calls appears to use reflections to register RPC function calls (and then dispatch them). In my language, this is done through the type system. Thus, we are able to constrain what possible operations can occur, and build such a system in a manner where fewer errors can occur (such as providing an incorrect function name). One major downside to this, however, is that, currently, the syntax for selecting what operation to perform from an internal choice is harder to read then specifying function names.

3. This section of code manages how the server processes RPC calls. In Go, much of this is abstracted away to a library—leaving much of the server code to describe how to start listening for connection and handling errors. In my language, there is less of a need for a library to perform these tasks and, as such, much of the code for handling the RPC calls is written here. This is done through using an ExternalChoie case on the channel. One downside

to this approach is that, the current syntax for case is somewhat hard to read as branchs are specified by order and separated by commas. My language's implementation is also limited to assuming that there will not be any connection errors.

D.5 Mobile Agents

Description

In the Mobile Agents paradigm, programs move between systems during execution in order to utilize the local resources of various systems (including compute power).

Code

Program 8: Java Mobile Agent Implementation [4]

```
1  import java.io.IOException;
2  import java.io.ObjectOutputStream;
3  import java.io.Serializable;
4  import java.net.Socket;
5  import java.util.LinkedList;  1
6
7  public abstract class Agent implements Runnable, Serializable {
8
9      public class HostInfo implements Serializable {
10         public String host;
11         public int port;
12
13         public HostInfo(String host, int port) {
14             this.host = host;
15             this.port = port;
16         }
17     }
18
19   private LinkedList<HostInfo> hosts;
20
21   public Agent() {
22     hosts = new LinkedList<HostInfo>();
23   }
24
25   public abstract void execute();  2
26
27   public void goTo(HostInfo h) {                                          3
28     try {
29       Socket socket = new Socket(h.host, h.port);
30
31       ObjectOutputStream output =                                         4
32   4                 new ObjectOutputStream(socket.getOutputStream());
33       output.writeObject(this);
34
35       socket.close();
36     } catch (IOException e)  {
37       // Handle Error
38     }
39   }
40
41   public void run() {
42     execute();
43
44     if(hosts.isEmpty()) return;
45     goTo(hosts.removeFirst());
46   }
47 }
```

Program 9: Bismuth Calculus Mobile Agent Client

```
1  define struct Host { ... }
2  define struct AgentData {
3      Host[] hosts;
4      ..... ———————— 1a
5  }
6
7  define AgentClient<P> :: c : Channel<+AgentData; P> = {
8      AgentData data = c.recv();
9      ....    ———————————————— 2
10
11     if data.hosts.length != 0 {
12         Channel<
13             -AgentData;
14             -Program<-AgentData;P>;
15             +Channel<not(P)>> channel = connect hosts.pop();
16         channel.send(data);
17         channel.send(AgentClient);
18         Channel<not(P)>> io = channel.recv();
19         ...
20      }
21 }
```

1

3

Program 10: Java Mobile Agent Server Implementation [4]

```java
import java.io.ObjectInputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class AgentServer {

    public static void main(String[] args) {
        try {
            ServerSocket socket = new ServerSocket(<port>);

            while(true) {
                Socket s = socket.accept();

                ObjectInputStream in = new ObjectInputStream(s.getInputStream());
                Agent a = (Agent) in.readObject();

                a.run();
            }

        } catch (Exception e) {
            // Handle Error
        }
    }
}
```

4

Program 11: Bismuth Calculus Mobile Agent Server

```
define RestrictedAgentServer :: c : Channel<!(
                                        +AgentData;
                                        +AgentClient;
                                        -Channel<not(P)>
                                    )> = {
    accept(c) {
        AgentData data = c.recv();
        AgentClient prog = c.recv();
        Channel<not(+AgentData;P)> channel = exec prog;
        channel.send(data);

        c.send(channel);
    }
}
define PermissiveAgentServer :: c : Channel<!(+Program<P>;-Channel<not(P)>)> = {
    accept(c) {
        Program<P> prog = c.recv();
        Channel<not(P)> channel = exec prog;
        c.send(channel);
    }
}
```

5

6

5

6

Comments

1. Definition of the mobile agent. In Java, this is done through an abstract class which provides the basic functionality for creating an agent that moves between devices. In my language,

lacking objects, this is done through a struct that holds the data needed by the agent, and another agent program which is executed and provided with the data it needs by the server.

(a) In Java, this makes it very easy for a developer to extend the Agent class to develop their own agent with unique functions. In my version, this is more challenging as the data that the agent uses must pass through the server. As such, the agents and agent data that my server is able to process is restricted when compared to the version in Java: instead of just extending the Agent class to create new agents, my language requires that the server also be informed of this change. This, however, could be seen as a benefit in that the server cannot execute arbitrary code. In addition, this could be worked around by making the agents use more permissive channels wherein a program with a polymorphic protocol is accepted instead of a specific program.

(b) In my language, it is much more challenging to establish inherited properties—thus meaning that each agent may have to define how it moves between systems. This could be solved by creating a wrapper agent program that has the code for managing these cases and then passing a program to this wrapper program to execute as the agent; however, this requires much more effort than the Java version where this can be accomplished through direct inheritance.

2. This is where the actual code of the mobile agent gets executed. In Java, this is a specific method which allows implementation of various Agents to inherit properties from the abstract definition. In my language, lacking an object oriented design, this becomes a block of code within each agent program's definition. One critical difference here is that, in Java, this block could be used to access resources local to the host that the program in running on. In my language, lacking a global state, it becomes unclear how such a functionality could be achieved as it would require the server to pass each client the resources relevant to it. In addition, even if there was a global state pointer of persistent resources passed to each program, now that programs are able to move, it may become unsafe to assume certain resources exist. This is a problem as currently the language has no means of error handling, and typechecking such programs could be complicated.

3. Enables agents to move between hosts. In both implementations, linear resources (such as file handles and channels) cannot be moved between hosts. In my version, however, they could be forwarded by sending these resources over the channel to the agent process. This is because, in my language, mobile agents can communicate with each other as they move between devices. In the Java implementation, this is not directly possible and would be somewhat challenging to implement.

4. Because Java does not have typed sessions, here we are forced to convert our Agent to an Object on the client, and an Object to an Agent on the server. These operations are security risks and could cause either the server to crash.

5. This demonstrates two potential sessions for a mobile agent server to use. In the first, only agents of the specific AgentClient program type will be accepted (this, however, could be expanded to more programs by using a sum type). This is fairly restrictive and thus makes it more challenging to achieve arbitrary code execution by sending a random program to the server. In addition, because we are given specific program types, we are also able to provide each program with the unique local resources it needs (however, this is not shown in the

current example). One challenge with this style, however, is that we do not currently have an easy way to describe a program being able to connect with both the server and previous agent client at once. In contrast, the permissive server is able to accept a wide variety of programs and provide their channel back to the previous agent. The downside here is that we would be unable to provide each program with specific data needed for its execution, and the system is less secure due to accepting any program. In addition, throughout both of these programs, due to the use of generic channels, we would need to define a syntactic element for inversing the types of channels. In the original calculus, this was done with the ¬ symbol; however, as this symbol does not appear on keyboards, we would have to use something else for an actual implementation. While ! is typically used to negate values in programming languages, this is currently used to represent loops of durations that are controlled by the other program. As such, we would need a different way to represent this. In addition, tracking the types of protocols through such operations can take some effort. This is because, when we execute a process, we get a channel that follows that follows the protocol that is inverse to the program; however, when we send and receive channels and programs themselves, these changes do not occur. As such, we have to plan ahead and think about how we will be obtaining channels in order to reason about what type of protocol they should correspond to.

6. Here, it is possible for us to be sending a completed channel. The problem with this is that there is an inference rule which deletes channels that have been completed. As such, typechecking these programs might be more challenging as we would need the resource to exist until here; however, we also need to be able to move the resource (and potentially destroy it once it is empty).

D.6 Quicksort

Description

Quicksort is a fast algorithm for sorting arrays which frequently runs subtasks concurrently to make the algorithm have an even shorter runtime.

Code

Program 12: C++ Quicksort

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <future>

using namespace std;

vector<int> quicksort(vector<int> input) {
    if(input.size() <= 1) return input;

    if(input.size() == 2)
    {
        if(input.at(0) > input.at(1))
        {
            int temp = input.at(0);
            input.erase(input.begin());
            input.push_back(temp);
        }
        return input;
    }

    int pivot = input.at(0);
    vector<int> less;
    vector<int> gtr;

    for(int i = 1; i < input.size(); i++)
    {
        int c = input.at(i);
        if(c < pivot) less.push_back(c);
        else gtr.push_back(c);
    }

    future<vector<int>> lhs = async(quicksort, less);
    future<vector<int>> rhs = async(quicksort, gtr);

    less = lhs.get();
    gtr = rhs.get();

    less.push_back(pivot);
    for(int i = 0; i < gtr.size(); i++)
    {
        less.push_back(gtr.at(i));
    }

    return less;
}
```

Program 13: Bismuth Calculus Quicksort

```
1  define QuickSort :: c : Channel<+int[]; -int[]> = {
2      int[] input := c.recv();
3
4      if input.length <= 1 {                                    1
5          c.send(input);
6          exit;                                                 2
7      }
8
9      if input.length == 2 {
10         if input[0] > input[1] {
11             int temp = input[0];
12             input[0] = input[1];
13             input[1] = temp;
14         }
15
16         c.send(input);
17         exit;
18     }
19
20     int current = input[0];
21     int[] less = new int[];
22     int[] gtr = new int[];
23
24     int i = 1;                                                3
25     while i < input.length {
26         if (input[i] < current) less.push(input[i]);                                  4
27         else gtr.push(input[i]);
28         i = i + 1;
29     }
30
31     var lhs = exec QuickSort;
32     var rhs = exec QuickSort;
33
34     lhs.send(less);
35     rhs.send(gtr);
36
37     less = lhs.recv();
38     gtr = rhs.recv();
39
40     less.push(current);
41
42     i = 0;
43
44     while i < gtr.length {
45         less.push(i);
46         i = i + 1;
47     }
48
49     c.send(less);
50 }
```

Comments

As the two implementations of quicksort are incredibly similar with the exception of their signatures and basic interactions with processes vs functions, this study will primarily focus on what writing this function in my language tells us about the language.

1. While my language does not have a rule to describe how if statements work, given their similarities to (and ability to be simulated by) external choices, we can assume that the two would work similarly.

2. Being able to typecheck statements like this become somewhat complicated as we have to verify that the linear uses of variables across each possible flow are possible—which is dependent on if each subprogram ends in an exit.

3. Here, my language's representation of the code is much less compact than the version in C++. This is because my language does not currently have a for loop; however, such a concept should be possible given the presense of rules for while loops.

4. Here, we use an object-oriented style of function on the arrays. While it would not be unreasonable to assume a similar kind of functionality would exist in my language, this does raise the question of how these concepts would be represented and if there should be some notion of objects in the language.

D.7 RDT

Description

Reliable Data Transmission (RDT) is a simple algorithm used to ensure that data is reliably transmitted across an unreliable medium. While such algorithms do have their limits, they are foundational for many networked systems and protocols such as TCP and QUIC which are currently responsible for handling all web traffic.

Code

Program 14: Rust Simple RDT Sender

```
use std::sync::mpsc::{Sender, Receiver};
use std::net::{SocketAddr, UdpSocket};
use std::time::{Duration, Instant};
use std::sync::mpsc::channel;
use std::thread;
static timeout: Duration = Duration::new(1, 0);

fn rdt_send(socket: UdpSocket, rdt_in: Receiver<u8>) {
    socket.set_read_timeout(Some(timeout));

    let mut seq_no = 0;

    thread::spawn(move|| {
        for msg in rdt_in {
            let sndpkt = make_pkt(seq_no, msg);

            socket.send(&sndpkt);

            let mut valid = false;
            let mut start = Instant::now();

            while !valid {
                while !valid && start.elapsed().lt(&timeout) {
                    let mut buf = [0; 10];
                    match socket.recv(&mut buf) {
                        Ok(received) => {
                            if is_valid(buf) {
                                valid = true;
                                seq_no = seq_no + 1;  //Note: Receiver omitted
                            }
                        },
                        Err(e) => {
                            //Handle Error
                        },
                    }
                }

                if !valid {
                    socket.send(&sndpkt);
                    start = Instant::now();
                }
            }
        }
    });
}
```

Program 15: Rust Simple RDT Receiver

```
use std::sync::mpsc::{Sender, Receiver};
use std::net::{SocketAddr, UdpSocket};
use std::time::{Duration, Instant};
```

```
use std::sync::mpsc::channel;
use std::thread;
static timeout: Duration = Duration::new(1, 0);
                    1

fn rdt_recv(socket: UdpSocket) -> Receiver<u8> {
    let (send, recv): (Sender<u8>, Receiver<u8>) = channel();                3

    thread::spawn(move|| {                                                   3
        let mut seq_no = 0;
        while true {
            let mut buf = [0; 10];
            match socket.recv(&mut buf) {
                Ok(received) => {
                    if is_valid(buf) {
                        send.send(extract(buf));
                        seq_no = seq_no + 1;
                    }
                },
                Err(e) => {
                    // Handle Error
                },
            }

            let sndpkt = make_pkt(seq_no, seq_no - 1); //Simulates ACK
            socket.send(&sndpkt);

        }
    });

    return recv;                                                             3
}
```

Program 16: Bismuth Calculus RDT Sender

```
1  define rdt_send :: c : ... = {
2      Channel<?-Packet> udt_out = ...;
3      Channel<!+Packet> udt_in = ...;
4
5      Channel<!+Data> rdt_in = ...;
6      Channel<?-Data> rdt_out = ...;
7      int seq_no = 0;
8
9      accept(rdt_in) {
10         Data d = rdt_in.recv();
11
12         Packet sndpkt = make_pkt(seq_no, data);
13
14         more(udt_out);
15         udt_out.send(sndpkt);
16
17         var time = ... ;
18         boolean valid = false;
19
20         while(!valid) {
21             acceptWhile(udt_in, udt_in.is_present()) {
22                 Packet p = udt_in.recv();
23
24                 valid = !corrupt(p) && isAck(p, seq_no);
25
26                 if(valid) {
27                     Data data = extract(p);
28                     more(rdt_out);
29                     rdt_out.send(data);
30
31                     seq_no = seq_no + 1;
32                 }
33             }
34
35             if (!valid && (... - time) > timeout) {
36                 more(udt_out);
37                 udt_out.send(sndpkt);
38                 time = ... ;
39             }
40         }
41     }
42
43     accept(udt_in) {
44         Packet recvpkt = udt_in.recv();
45     }
46
47     weaken(udt_out);
48     weaken(rdt_out);
49 }
```

1
2
4
4
4
5

Program 17: Bismuth Calculus RDT Receiver

```
1  define rdt_recv :: c : ... = {                                    5
2      Channel<?-Packet> udt_out = ...;
3      Channel<!+Packet> udt_in = ...;                             } 1
4
5      Channel<?-Data> rdt_out = ...;                                    3
6
7      int seq_no = 0;
8      accept(udt_in) {
9          Packet recvpkt = udt_in.recv();
10
11         if(!corrupt(recvpkt) && has_seq(seq_no, recvpkt)) {
12             Data data = extract(recvpkt);
13             more(rdt_out);
14             rdt_out.send(data);
15
16             seq_no = seq_no + 1;
17         }
18
19         Packet sndpkt = make_packet(ACK, seq_no - 1);
20         more(udt_out);
21         udt_out.send(sndpkt);
22     }
23
24     weaken(udt_out);
25     weaken(rdt_out);
26 }
```

Comments

1. Socket to unreliable data transmission. In Rust, this is managed in one variable. In my language, this is done in two as we need one channel for each direction of communication.

2. Here, the RDT sender has to receive packets from the client to verify that the message was received. If this does not occur within a certain timeout, the sender needs to resend packet. In Rust, this can be done fairly easily as it is possible to set a timeout on channels; however, in my language, due to channels being linear, this is much more challenging. To accomplish it, we are required to accept data from the channel only when data is present so that way we can track our timeout without blocking—two features which are not in the initial language definition. This approach, however, is incredibly inefficient as it serves as a *spinlock* (in part, due to our inability to race processes to see which completes first). In addition, having to resend packets also means that it is impossible for us to implement RDT for linear types. One potential solution for this is to consider unreliable data transmission mechanisms to be non-linear, and then to have reliable data transmission be implemented within the language's runtime, and allow that to communicate linear data.

3. Depending on how the RDT system is designed, we sometimes need to create a channel and return that to another process. In Rust, this highlights the need to have threads as we will need a way to have computation continue after returning the channel to the other process. In my language, processes can continue to execute after having completed their channel requirements—thus enabling us to write code in a more expressive manner where we

can think of having already established a communication channel with another process before going to use it.

4. This combination of nested accepts, whiles and (the hypothetical feature) accept whiles suggests that we would need loops to function in a different way than the initial language's rules suggest. In the current rules, it is sometimes challenging to bring loops into other loops due to the potential to weaken the loop—thus causing the program to enter an invalid state on the next iteration. The need for such a pattern here, though, confirms that the language needs to have a way to handle such cases.

5. Having channel types both helps the development process and makes it more challenging. Once we have a channel for a program, we are able to learn a lot about what kind of structure the program needs in order to recognize the channel; however, the process of determining what this channel should be can be quite challenging. For instance, we have to determine if we want the process to recognize a channel, or if the process primarily interacts with processes that it executes/connects to (such as in this example). Other challenges in determining the protocol include:

   (a) resolving mismatches between how we use natural language and math - for instance, if we have a process where we are given the option to perform a task or move on to a different task, and, every time we complete the current task, we must make the decision of if we continue to repeat the current task or move onto the next one, it may seem natural that this process would be described with an internal choice. This, however, is incorrect; instead, we would simply be given a loop that we control the duration of then, whenever we decide that the loop is over, we step to the next task. As such, in this case, developers may tend to conflate a sequential step with an internal choice—similarly to how natural language may make one conflate 'and' and 'or' when learning boolean logic.

   (b) extrapolating a two-sided protocol from a flowchart which lacks such context - in writing protocols, we have to specify the differences between operations such as internal/external choice and of course/why not. In flowcharts (and other conceptualizations of protocols), these concepts are usually reduced to an or and a loop. As such, the developer may be required to extrapolate this information, and, given the complexities of protocols and the ability to create multiple different representations of the same protocol, this may be challenging.

   (c) reducing complexities into a simplistic protocol language - there are many kinds of protocols for which this language cannot yet describe (such as the case of a protocol aborting). As such, there may be times when it is unclear what operation a protocol should use as both processes seem to exert some degree of control over the program's functionality when such a concept cannot be encoded into the current language. In addition, having to map channels between representations can be a very confusing and computationally expensive process.

D.8 IsPrime

Description

IsPrime is a simple function that accepts an integer and returns true if the integer is prime and false otherwise.

Code

Program 18: Standard IsPrime Implementation

```
1  boolean isPrime(int n) {
2      int i = 3;
3
4      while (i < n) {
5          if (n / i * i = n) {
6              return false;
7          }
8          i := i + 2;
9      }
10
11     return true;
12 }
```

Program 19: Bismuth Calculus - IsPrime Incorrect Implementation

```
1  define isPrime :: c : Channel<+int;-boolean> = {
2      int n = c.recv();
3      int i = 3;
4
5      while (i < n) {
6          if (n / i * i = n) { c.send(false) exit; }
7          i := i + 2;
8      }
9      c.send(true);
10 }
```

Program 20: Bismuth Calculus - IsPrime Implementation

```
1  define isPrime :: c : Channel<+int;-boolean> = {
2      int n = c.recv();
3      int i = 3;
4
5      boolean done = false;
6      boolean ans = true;
7
8      while (!done && i < n) {
9          if (n / i * i = n)
10         {
11             done = true;
12             ans = false;
13         }
14         else
15         {
16             i := i + 2;
17         }
18     }
19
20     c.send(ans);
21 }
```

2

1

Comments

1. While this program may initially appear correct due to its similarities (both in structure and logic) to the traditional implementation of this algorithm, the program here is actually invalid as, currentlty, the channel c would have to be of type ?boolean in order to be accessible within loops. This is because, in a traditional function, a return statement marks the end of a function—and thus we can only ever reach one return statement in a function. In contrast, channels can be communicated across any number of times (as recognized by the channel's protocol)—preventing us from being able to bring channels for which we do not have control over their repetition into loops as otherwise we could have a protocol violation.

2. In the traditional implementation, function parameters are named which can help give a meaning to what the function does. Not only do we not have this with protocols, but then we have to separately receive each value. This can make communicating between processes seem more bulky than channels as more steps are involved. This violates two principles of the Cognitive Dimensions framework:

   (a) Secondary Notation - By eliminating the ability to name parameters, we erase a secondary means for developers to communicate.

   (b) Viscosity - By requiring the channels to both define what operations the program will do, and then requiring the programmer to write these steps, we make changes more difficult as they now have to edit more code.

E An Argument for Protocol Types as Regular Expressions

In many of today's popular languages (C, C++, Java, etc), types serve as a restriction on the language that allows us to gain information about the resource that they describe. This allows us to stipulate conditions under which our code is allowed to function. For instance, if I write a function that accepts two integers, not only can I assume that I have two integers within the function's definition, but it also requires that someone provides me with exactly two integers. In some respects, this is comparable to a descriptive statement (and thus lends to the notion that writing is the last step of a process and it simply documents the facts).

In contrast, in session typed languages, the types take on a much more active meaning: instead of describing something that must be true (as a requisite for the program to function), they take on much more of a performative role by enacting what the program must do. In this respect, our types evolve into a language of their own, and our code becomes a program which recognizes this language.

This also means that it is now critical for our types to be representative of the kinds of metaphors humans use in communicating processes. This, however, is where I believe many of these languages struggle. Many languages provide us with the basic mathematical building blocks from which we may construct more complex programs and not human metaphors. This is comparable to assembly language which, despite providing us the basic components that we build all other languages off of, is rarely used due to the fact that, regardless of how much abstraction one can achieve in the language, we are unable to leverage human metaphors to achieve a level of rhetoric and communication that higher-level languages can.

In the context of session types, this means that programmers must dedicate an excessive amount of time to programming the types instead of their code. This can be a time consuming logical puzzle in finding ways of representing human concepts in a terse mathematical language for which symbols sometimes take on different meanings depending on context. As our code must recognize the language of our session type, it too gets dragged into this complexity—thus limiting its human-readability.

This, however, is unnecessary given that, as demonstrated by higher-level languages, we can transform human metaphors into machine code. Thus, why not build session types on human metaphors instead? To do this, we must answer the question of how do people, specifically within the context of programming, define protocols?

For this, I believe an incredibly relevant answer is that of flow charts. This is because flowcharts are incredibly common design tools for designing and planning programs. In addition, flowcharts, by definition, are finite state machines and thus regular languages—for which we already have a standard way of representing textually and a mathematical definition for (the computation between and within each state becomes the code of the language). As such, we know that the basic blocks of our language's types must include the ability to represent a state, an option between states, and the ability to repeat states. This, however, only takes into account one side of the communication. As such, we must extend each of these operations into a version used by the local machine, and the one used by the peer. Logically, this means that state becomes send and receive, option becomes internal choice and external choice, and repetition becomes controlled by us or controlled by our peer. Finally, to overcome the limit that state machines are sequential, we must add an option which allows for parallel composition.

While this does not establish how these concepts should be represented in our language (how they should be communicated), they do go towards establishing what devices/metaphors have come

to be expected within the realm of CS in defining protocols.

F Refined Language Definition

F.1 General Definitions

1. Equivalences:

$$\begin{array}{lcl}
+\tau & \equiv & \neg(-\tau) \\
\neg?\rho & \equiv & !(\neg\rho) \\
\neg(\rho;\rho) & \equiv & \neg\rho;\neg\rho \\
\neg(\rho \oplus \rho) & \equiv & \neg\rho \,\&\, \neg\rho \\
\lceil^{\bullet\bullet}\rho\rceil & \equiv & \lceil^{\bullet}\rho\rceil \\
\lceil(\rho_1 \oplus \rho_2);\rho_3\rceil & \equiv & \lceil(\rho_1;\rho_3) \oplus (\rho_2;\rho_3)\rceil \\
\lceil(\rho_1 \,\&\, \rho_2);\rho_3\rceil & \equiv & \lceil(\rho_1;\rho_3) \,\&\, (\rho_2;\rho_3)\rceil
\end{array}$$

2. Define $\Delta\{e\}$ such that $(\Delta, x:\tau)\{x\} = \Delta$ and $(\Delta)\{x\} = \Delta$.

3. Environment Insert Operator:

$$(\Gamma;\Delta)\langle x\rangle = \begin{cases} \Gamma;\Delta,x & \text{if } x : \lceil\rho\rceil \\ \Gamma,x;\Delta & \text{if } x:\tau \end{cases}$$

4. Buffer Function:

$$\Sigma_B^G(\rho)(c) = \begin{cases} \rho & \text{if } G(c) \equiv \epsilon \\ +\tau;\Sigma_B^{G[c\mapsto q']}(\rho)(c) & \text{if } G(c) \equiv v[\tau];q' \\ (\Sigma_B^{G[c\mapsto q']}(\rho)(c)) \,\&\, B & \text{if } G(c) \equiv \mathsf{SEL}[\cdot \,\&\, B];q' \\ A \,\&\, (\Sigma_B^{G[c\mapsto q']}(\rho)(c)) & \text{if } G(c) \equiv \mathsf{SEL}[A \,\&\, \cdot];q' \\ \text{let } \alpha;!\alpha;\beta = \Sigma_B^{G[c\mapsto q']}(\rho)(c) \text{ in } !\alpha;\beta & \mathsf{START{-}LOOP};q' \\ !A;\Sigma_B^{G[c\mapsto q']}(\rho)(c) & \text{if } G(c) \equiv \mathsf{END{-}LOOP}[!A] \end{cases}$$

F.2 Send

$$\text{(TSend)} \quad \frac{\Gamma;\Delta \vdash e:\tau \quad \Gamma;\Delta\{e\}, c:\lceil\rho\rceil \vdash P}{\Gamma;\Delta, c:\lceil\neg\tau;\rho\rceil \vdash c\langle e\rangle.P}$$

$$\text{(OPSend)} \quad (G; L_i; c\langle e\rangle.P) \mapsto (G'; L_i\{e\}; P) \qquad \begin{pmatrix} (G;L_i) \vdash e \Downarrow v \\ G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m} \\ G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, v \end{pmatrix}$$

F.3 Receive

$$\text{(TRecv)} \quad \frac{\Gamma\langle x\rangle;\Delta\langle x\rangle, c:\lceil\rho\rceil \vdash P}{\Gamma;\Delta, c:\lceil+\tau;\rho\rceil \vdash c(x).P}$$

(OPRecv) $(G; L_i; c(x).P) \mapsto (G'; L_i[x \mapsto v]; P)$ $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, c \mapsto v, \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m} \end{pmatrix}$

F.4 External Choice (Offer)

(TCase) $\dfrac{\Gamma; \Delta, c : \restriction \rho_1 \restriction \vdash P.R \quad \Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash Q.R}{\Gamma; \Delta, c : \restriction \rho_1 \mathbin{\&} \rho_2 \restriction \vdash c.\texttt{case}(P,\ Q).R}$

(OPCase1) $(G; L_i; c.\texttt{case}(P,\ Q).R) \mapsto (G'; L_i; P.R)$ $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{SEL}[\cdot \mathbin{\&} \rho_2], \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m} \end{pmatrix}$

(OPCase2) $(G; L_i; c.\texttt{case}(P,\ Q).R) \mapsto (G'; L_i; Q.R)$ $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{SEL}[\rho_1 \mathbin{\&} \cdot], \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m} \end{pmatrix}$

F.5 Internal Choice (Project/Select)

(TSelect1) $\dfrac{\Gamma; \Delta, c : \restriction \rho_1 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \rho_1 \oplus \rho_2 \restriction \vdash c[1].P}$

(OPSelect1) $(G; L_i; c[1].P) \mapsto (G'; L_i; P)$ $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[\cdot \mathbin{\&} \rho_2] \end{pmatrix}$

(TSelect2) $\dfrac{\Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \rho_1 \oplus \rho_2 \restriction \vdash c[2].P}$

(OPSelect2) $(G; L_i; c[2].P) \mapsto (G'; L_i; P)$ $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[\rho_1 \mathbin{\&} \cdot] \end{pmatrix}$

F.6 While Loop

(TWhile) $\dfrac{\Gamma; \cdot \vdash e : \mathsf{bool} \quad \Gamma; \Delta_1 \vdash P \quad \Gamma; \Delta \vdash Q}{\Gamma; \Delta \vdash \texttt{while}(e)\{P\}.Q}$ [1]

(OPWhileT) $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; P.\texttt{while}(e)\{P\}.Q)$ [2]

(OPWhileF) $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; Q)$ [3]

---

[1] $(\Delta_1 \equiv \{x_i : \restriction^{\bullet} \rho_i \restriction \text{ for all } x_i \in \Delta\})$
[2] $((G; L_i) \vdash e \Downarrow \mathsf{true})$
[3] $((G; L_i) \vdash e \Downarrow \mathsf{false})$

F.7 Unfold

$$\text{(T?Unfold)} \quad \frac{\Gamma; \Delta, c : \restriction \rho_1; ?\rho_1; \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction ?\rho_1; \rho_2 \restriction \vdash \texttt{more}(c).P}$$

$$\text{(T?Unfold Guarded)} \quad \frac{\Gamma; \Delta, c : \restriction \rho_1; \bullet ?\rho_1; \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \bullet ?\rho_1; \rho_2 \restriction \vdash \texttt{more}(c).P}$$

$$\text{(OP?Unfold)} \quad (G; L_i; \texttt{more}(c).P) \mapsto (G'; L_i; P) \quad [1]$$

[1] $\left(\begin{array}{l} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c \mapsto m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c \mapsto m}, \mathsf{START{-}LOOP} \end{array}\right)$

F.8 Weaken

$$\text{(T?Weaken)} \quad \frac{\Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction ?\rho_1; \rho_2 \restriction \vdash \texttt{weaken}(c).P}$$

$$\text{(OP?Weaken)} \quad (G; L_i; \texttt{weaken}(c).P) \mapsto (G'; L_i; P) \quad [1]$$

[1] $\left(\begin{array}{l} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c \mapsto m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overrightarrow{c \mapsto m}, \mathsf{END{-}LOOP}[!\neg\rho_1] \end{array}\right)$

F.9 Accept

$$\text{(TAccept)} \quad \frac{\Gamma; \Delta_1, c : \restriction \rho_1 \restriction \vdash P \quad \Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash Q}{\Gamma; \Delta, c : \restriction !\rho_1; \rho_2 \restriction \vdash \texttt{accept}(c)\{P\}.Q} \quad [1]$$

$$\text{(OPAccept-Loop)} \quad (G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{accept}(c)\{P\}.Q) \quad [2]$$

$$\text{(OPAccept-End)} \quad (G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; Q) \quad [3]$$

[1] $(\Delta_1 \equiv \{x_i : \restriction \bullet \rho_i \restriction \text{ for all } x_i \in \Delta\})$

[2] $\left(\begin{array}{l} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{array}\right)$

[3] $\left(\begin{array}{l} G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m} \end{array}\right)$

F.10 Accept While

$$\text{(TAcceptWhile)}\quad \frac{\Gamma;\cdot \vdash e : \mathsf{bool} \quad \Gamma;\Delta_1, c : \uparrow\rho_1\uparrow \vdash P \quad \Gamma;\Delta, c : \uparrow!\rho_1;\rho_2\uparrow \vdash Q}{\Gamma;\Delta, c : \uparrow!\rho_1;\rho_2\uparrow \vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q}\ {}^{1}$$

$$\text{(TAcceptWhile Guarded)}\quad \frac{\Gamma;\cdot \vdash e : \mathsf{bool} \quad \Gamma;\Delta_1, c : \uparrow\rho_1\uparrow \vdash P \quad \Gamma;\Delta, c : \uparrow^{\bullet}!\rho_1;\rho_2\uparrow \vdash Q}{\Gamma;\Delta, c : \uparrow^{\bullet}!\rho_1;\rho_2\uparrow \vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q}\ {}^{2}$$

$$\text{(OPAcceptWhile-Loop)}\quad (G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{acceptWhile}(c,\ e)\{P\}.Q)\ {}^{3}$$

$$\text{(OPAcceptWhile-Preempt)}\quad (G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)\ {}^{4}$$

$$\text{(OPAcceptWhile-End)}\quad (G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)\ {}^{5}$$

---

1 $(\Delta_1 \equiv \{x_i : \uparrow^{\bullet}\rho_i\uparrow \text{ for all } x_i \in \Delta\})$

2 $(\Delta_1 \equiv \{x_i : \uparrow^{\bullet}\rho_i\uparrow \text{ for all } x_i \in \Delta\})$

3 $\begin{pmatrix}(G; L_i) \vdash e \Downarrow \mathsf{true} \\ G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m} \\ G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}\end{pmatrix}$

4 $\begin{pmatrix}(G; L_i) \vdash e \Downarrow \mathsf{false} \\ G' \equiv G\end{pmatrix}$

5 $\begin{pmatrix}G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m} \\ G' \equiv G\end{pmatrix}$

F.11 Close Channel

$$\text{(TClose)}\quad \frac{\Gamma;\Delta \vdash P}{\Gamma;\Delta, c : \uparrow 1\uparrow \vdash \texttt{close}(c).P}$$

$$\text{(OPClose)}\quad (L_i; \texttt{close}(c).P) \mapsto (L_i; P) \qquad \begin{pmatrix}G' \equiv G \\ L_i' \equiv L_i\end{pmatrix}$$

F.12 Close Process

$$\text{(TExit)}\quad \frac{}{\Gamma;\cdot \vdash \texttt{exit}}$$

$$\text{(OPExit-Continue)}\quad (G; \overrightarrow{L}; (P_1, \ldots, \texttt{exit}, \ldots, P_n)) \mapsto (G; \overrightarrow{L}; (P_1, \ldots, P_n))$$

$$\text{(OPExit-Isolated)}\quad (G; \overrightarrow{L}; \texttt{exit})\ \mathsf{done}$$

F.13 Skip

$$\text{(TSkip)} \quad \frac{}{\Gamma; \Delta \vdash \texttt{skip}} \ {}^{1}$$

$$\text{(OPSkip-Continue)} \quad (G; L_i; \texttt{skip}.P) \mapsto (G; L_i; P)$$

$$\text{(OPSkip-Exit)} \quad (G; L_i; \texttt{skip}) \mapsto (G; L_i; \texttt{exit})$$

[1] (For all $x \in \Delta$,$\Delta(x) = \uparrow^{\bullet} \rho \uparrow$)

F.14 Execute Program

$$\text{(TExec-Prog)} \quad \frac{\Gamma; \Delta, c : \uparrow \neg \rho \uparrow \vdash Q \quad \cdot; n : \uparrow \rho \uparrow \vdash P}{\Gamma; \Delta \vdash (c = \texttt{exec}\ \ P :: n : \rho).Q}$$

$$\text{(OPExec-Prog)} \quad (G; L_i; (c = \texttt{exec}\ \ P :: n : \rho).Q) \mapsto (G'; (L'_i, L'_P); (P, Q)) \quad {}^{1}$$

---

1 $\begin{pmatrix} G \equiv \overrightarrow{z \mapsto q} \\ G' \equiv \overrightarrow{z \mapsto q}, fn_c \mapsto [], fn_n \mapsto [] \\ L'_i \equiv L_i[c \mapsto fn_c] \\ L'_P \equiv \{n \mapsto fn_n\} \\ \overline{c} = n \\ \overline{n} = c \end{pmatrix}$

G Progress Proof

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

Case 1: (TSend) $\dfrac{\Gamma; \Delta \vdash e : \tau \quad \Gamma; \Delta\{e\}, c : \uparrow\rho\uparrow \vdash P}{\Gamma; \Delta, c : \uparrow\neg\tau; \rho\uparrow \vdash c\langle e\rangle.P}$

(a) By case assumption for TSend, P has form $c\langle e\rangle.Q$. As such, OPSend applies.

(b) By OPSend, $(G; L_i; c\langle e\rangle.P) \mapsto (G'; L_i\{e\}; P)$ where:

- $(G; L_i) \vdash e \Downarrow v$
- $G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, v$

(c) Thus, by OPUniversalStep, the entire system steps.

Case 2: (TRecv) $\dfrac{\Gamma\langle x\rangle; \Delta\langle x\rangle, c : \uparrow\rho\uparrow \vdash P}{\Gamma; \Delta, c : \uparrow+\tau; \rho\uparrow \vdash c(x).P}$

(a) By case assumption for TRecv, P has form $c(x).Q$. As such, OPRecv applies.

(b) By OPRecv, $(G; L_i; c(x).P) \mapsto (G'; L_i[x \mapsto v]; P)$ where:

- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto v, \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

(c) Thus, by OPUniversalStep, the entire system steps.

Case 3: (TCase) $\dfrac{\Gamma; \Delta, c : \uparrow\rho_1\uparrow \vdash P.R \quad \Gamma; \Delta, c : \uparrow\rho_2\uparrow \vdash Q.R}{\Gamma; \Delta, c : \uparrow\rho_1 \,\&\, \rho_2\uparrow \vdash c.\mathtt{case}(P,\ Q).R}$

- **Lemma 1:** Either $\Sigma_B^G(c) = \mathsf{SEL}[\cdot \,\&\, \rho_2], \overrightarrow{m}$ or $\Sigma_B^G(c) = \mathsf{SEL}[\rho_1 \,\&\, \cdot], \overrightarrow{m}$.

(a) If $\Sigma_B^G(c) = \mathsf{SEL}[\cdot \,\&\, \rho_2], \overrightarrow{m}$:

i. By case assumption for TCase, P has form $c.\mathtt{case}(Q,\ R).S$ and $\Sigma_B^G(c) = \mathsf{SEL}[\cdot \,\&\, \rho_2], \overrightarrow{m}$. As such, OPCase1 applies.

ii. By OPCase1, $(G; L_i; c.\mathtt{case}(P,\ Q).R) \mapsto (G'; L_i; P.R)$ where:

- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{SEL}[\cdot \,\&\, \rho_2], \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Thus, by OPUniversalStep, the entire system steps.

(b) If $\Sigma_B^G(c) = \mathsf{SEL}[\rho_1 \,\&\, \cdot], \overrightarrow{m}$:

i. By case assumption for TCase, P has form $c.\mathtt{case}(Q,\ R).S$ and $\Sigma_B^G(c) = \mathsf{SEL}[\rho_1 \,\&\, \cdot], \overrightarrow{m}$. As such, OPCase2 applies.

ii. By OPCase2, $(G; L_i; c.\mathtt{case}(P,\ Q).R) \mapsto (G'; L_i; Q.R)$ where:

- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{SEL}[\rho_1 \,\&\, \cdot], \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 4: Internal Choice

Case 4a: (TSelect1) $\dfrac{\Gamma; \Delta, c : \restriction \rho_1 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \rho_1 \oplus \rho_2 \restriction \vdash c[1].P}$

i. By case assumption for TSelect1, P has form $c[1].Q$. As such, OPSelect1 applies.

ii. By OPSelect1, $(G; L_i; c[1].P) \mapsto (G'; L_i; P)$ where:

- $G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, \mathsf{SEL}[\cdot \,\&\, \rho_2]$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 4b: (TSelect2) $\dfrac{\Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \rho_1 \oplus \rho_2 \restriction \vdash c[2].P}$

i. By case assumption for TSelect2, P has form $c[2].Q$. As such, OPSelect2 applies.

ii. By OPSelect2, $(G; L_i; c[2].P) \mapsto (G'; L_i; P)$ where:

- $G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, \mathsf{SEL}[\rho_1 \,\&\, \cdot]$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 5: (TWhile) $\dfrac{\Gamma; \cdot \vdash e : \mathsf{bool} \quad \Gamma; \Delta_1 \vdash P \quad \Gamma; \Delta \vdash Q}{\Gamma; \Delta \vdash \texttt{while}(e)\{P\}.Q} \qquad \left(\Delta_1 \equiv \{x_i : \restriction^{\bullet} \rho_i \restriction \text{ for all } x_i \in \Delta\}\right)$

- **Lemma 1:** $(G; L_i) \vdash e \Downarrow v$ and $v : \mathsf{bool}$. By canonical forms, v is true or v is false.

(a) If v is true:

i. By case assumption for TWhile, P has form $\texttt{while}(e)\{Q\}.R$ and $e \Downarrow \mathsf{true}$. As such, OPWhileT applies.

ii. By OPWhileT, $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; P.\texttt{while}(e)\{P\}.Q)$ where:
- $(G; L_i) \vdash e \Downarrow \mathsf{true}$

iii. Thus, by OPUniversalStep, the entire system steps.

(b) If v is false:

i. By case assumption for TWhile, P has form $\texttt{while}(e)\{Q\}.R$ and $e \Downarrow \mathsf{false}$. As such, OPWhileF applies.

ii. By OPWhileF, $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; Q)$ where:
- $(G; L_i) \vdash e \Downarrow \mathsf{false}$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 6: (T?Unfold) $\dfrac{\Gamma; \Delta, c : \restriction \rho_1; ?\rho_1; \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction ?\rho_1; \rho_2 \restriction \vdash \texttt{more}(c).P}$

(a) By case assumption for T?Unfold, P has form $\texttt{more}(c).Q$. As such, OP?Unfold applies.

(b) By OP?Unfold, $(G; L_i; \texttt{more}(c).P) \mapsto (G'; L_i; P)$ where:

- $G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c \mapsto m}, \mathsf{START{-}LOOP}$

(c) Thus, by OPUniversalStep, the entire system steps.

Case 7: (T?Unfold Guarded) $\dfrac{\Gamma; \Delta, c : \restriction \rho_1; \bullet ?\rho_1; \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction \bullet ?\rho_1; \rho_2 \restriction \vdash \texttt{more}(c).P}$

(a) By case assumption for T?Unfold Guarded, P has form $\texttt{more}(c).Q$. As such, OP?Unfold applies.

(b) By OP?Unfold, $(G; L_i; \texttt{more}(c).P) \mapsto (G'; L_i; P)$ where:

- $G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overline{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{START{-}LOOP}$

(c) Thus, by OPUniversalStep, the entire system steps.

Case 8: (T?Weaken) $\dfrac{\Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash P}{\Gamma; \Delta, c : \restriction ?\rho_1; \rho_2 \restriction \vdash \texttt{weaken}(c).P}$

(a) By case assumption for T?Weaken, P has form $\texttt{weaken}(c).Q$. As such, OP?Weaken applies.

(b) By OP?Weaken, $(G; L_i; \texttt{weaken}(c).P) \mapsto (G'; L_i; P)$ where:

- $G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overline{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{END{-}LOOP}[!\neg\rho_1]$

(c) Thus, by OPUniversalStep, the entire system steps.

Case 9: (TAccept) $\dfrac{\Gamma; \Delta_1, c : \restriction \rho_1 \restriction \vdash P \quad \Gamma; \Delta, c : \restriction \rho_2 \restriction \vdash Q}{\Gamma; \Delta, c : \restriction !\rho_1; \rho_2 \restriction \vdash \texttt{accept}(c)\{P\}.Q} \qquad \left(\Delta_1 \equiv \{x_i : \restriction \bullet \rho_i \restriction \text{ for all } x_i \in \Delta\}\right)$

- **Lemma 1:** Either $\Sigma_B^G(c) = \mathsf{START{-}LOOP}, \overrightarrow{m}$ or $\Sigma_B^G(c) = \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$.

(a) If $\Sigma_B^G(c) = \mathsf{START{-}LOOP}, \overrightarrow{m}$:

i. By case assumption for TAccept, P has form $\texttt{accept}(c)\{Q\}.R$ and $\Sigma_B^G(c) = \mathsf{START{-}LOOP}, \overrightarrow{m}$. As such, OPAccept-Loop applies.

ii. By OPAccept-Loop, $(G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{accept}(c)\{P\}.Q)$ where:

- $G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m}$

iii. Thus, by OPUniversalStep, the entire system steps.

(b) If $\Sigma_B^G(c) = \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$:

i. By case assumption for TAccept, P has form $\texttt{accept}(c)\{Q\}.R$ and $\Sigma_B^G(c) = \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$. As such, OPAccept-End applies.

ii. By OPAccept-End, $(G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; Q)$ where:

- $G \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto \overrightarrow{q}}, c \mapsto \overrightarrow{m}$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 10: (TAcceptWhile) $\dfrac{\Gamma;\cdot\vdash e:\mathsf{bool}\quad \Gamma;\Delta_1,c:\uparrow\rho_1\uparrow\vdash P\quad \Gamma;\Delta,c:\uparrow!\rho_1;\rho_2\uparrow\vdash Q}{\Gamma;\Delta,c:\uparrow!\rho_1;\rho_2\uparrow\vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q}$

- **Lemma 1:** Either $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{true}$; $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{false}$; or $\Sigma_B^G(c)=\mathsf{END{-}LOOP}[!\rho_1],\overrightarrow{m}$.

(a) If $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{true}$:

i. By case assumption for TAcceptWhile, P has form $\texttt{acceptWhile}(c,\ e)\{P\}.Q$ and $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{true}$. As such, OPAcceptWhile-Loop applies.

ii. By OPAcceptWhile-Loop, $\begin{array}{l}(G;L_i;\texttt{acceptWhile}(c,\ e)\{P\}.Q)\mapsto\\(G';L_i;P.\texttt{acceptWhile}(c,\ e)\{P\}.Q)\end{array}$ where:

- $(G;L_i)\vdash e\Downarrow\mathsf{true}$
- $G\equiv\overrightarrow{z\mapsto q},c\mapsto\mathsf{START{-}LOOP},\overrightarrow{m}$
- $G'\equiv\overrightarrow{z\mapsto q},c\mapsto\overrightarrow{m}$

iii. Thus, by OPUniversalStep, the entire system steps.

(b) If $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{false}$

i. By case assumption for TAcceptWhile, P has form $\texttt{acceptWhile}(c,\ e)\{P\}.Q$ and $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{false}$. As such, OPAcceptWhile-Preempt applies.

ii. By OPAcceptWhile-Preempt, $(G;L_i;\texttt{acceptWhile}(c,\ e)\{P\}.Q)\mapsto(G';L_i;Q)$ where:

- $(G;L_i)\vdash e\Downarrow\mathsf{false}$
- $G'\equiv G$

iii. Thus, by OPUniversalStep, the entire system steps.

(c) If $\Sigma_B^G(c)=\mathsf{END{-}LOOP}[!\rho_1],\overrightarrow{m}$:

i. By case assumption for TAcceptWhile, P has form $\texttt{acceptWhile}(c,\ e)\{P\}.Q$ and $\Sigma_B^G(c)=\mathsf{END{-}LOOP}[!\rho_1],\overrightarrow{m}$. As such, OPAcceptWhile-End applies.

ii. By OPAcceptWhile-End, $(G;L_i;\texttt{acceptWhile}(c,\ e)\{P\}.Q)\mapsto(G';L_i;Q)$ where:

- $G\equiv\overrightarrow{z\mapsto q},c\mapsto\mathsf{END{-}LOOP}[!\rho_1],\overrightarrow{m}$
- $G'\equiv G$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 11: (TAcceptWhile Guarded) $\dfrac{\Gamma;\cdot\vdash e:\mathsf{bool}\quad \Gamma;\Delta_1,c:\uparrow\rho_1\uparrow\vdash P\quad \Gamma;\Delta,c:\uparrow^\bullet!\rho_1;\rho_2\uparrow\vdash Q}{\Gamma;\Delta,c:\uparrow^\bullet!\rho_1;\rho_2\uparrow\vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q}$

- **Lemma 1:** Either $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{true}$; $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{false}$; or $\Sigma_B^G(c)=\mathsf{END{-}LOOP}[!\rho_1],\overrightarrow{m}$.

(a) If $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{true}$:

i. By case assumption for TAcceptWhile Guarded, P has form $\texttt{acceptWhile}(c,\ e)\{P\}.Q$ and $\Sigma_B^G(c)=\mathsf{START{-}LOOP},\overrightarrow{m}$ and $(G;L_i)\vdash e\Downarrow\mathsf{true}$. As such, OPAcceptWhile-Loop applies.

ii. By OPAcceptWhile-Loop, $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{acceptWhile}(c,\ e)\{P\}.Q)$ where:
- $(G; L_i) \vdash e \Downarrow \mathsf{true}$
- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Thus, by OPUniversalStep, the entire system steps.

(b) If $\Sigma_B^G(c) = \mathsf{START{-}LOOP}, \overrightarrow{m}$ and $(G; L_i) \vdash e \Downarrow \mathsf{false}$

i. By case assumption for TAcceptWhile Guarded, P has form $\texttt{acceptWhile}(c,\ e)\{P\}.Q$ and $\Sigma_B^G(c) = \mathsf{START{-}LOOP}, \overrightarrow{m}$ and $(G; L_i) \vdash e \Downarrow \mathsf{false}$. As such, OPAcceptWhile-Preempt applies.

ii. By OPAcceptWhile-Preempt, $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)$ where:
- $(G; L_i) \vdash e \Downarrow \mathsf{false}$
- $G' \equiv G$

iii. Thus, by OPUniversalStep, the entire system steps.

(c) If $\Sigma_B^G(c) = \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$:

i. By case assumption for TAcceptWhile Guarded, P has form $\texttt{acceptWhile}(c,\ e)\{P\}.Q$ and $\Sigma_B^G(c) = \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$. As such, OPAcceptWhile-End applies.

ii. By OPAcceptWhile-End, $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)$ where:
- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$
- $G' \equiv G$

iii. Thus, by OPUniversalStep, the entire system steps.

Case 12: (TClose) $\dfrac{\Gamma; \Delta \vdash P}{\Gamma; \Delta, c : \uparrow 1 \uparrow \vdash \texttt{close}(c).P}$ ()

(a) By case assumption for TClose, P has form P. As such, OPClose applies.

(b) By OPClose, $(L_i; \texttt{close}(c).P) \mapsto (L_i; P)$ where:
- $G' \equiv G$
- $L_i' \equiv L_i$

(c) Thus, by OPUniversalStep, the entire system steps.

Case 13: (TExit) $\dfrac{}{\Gamma; \cdot \vdash \texttt{exit}}$

(a) If P_i is the last process running:

i. By case assumption for TExit, P has form `exit` and P_i is the last process running. As such, OPExit-Isolated applies.

ii. By OPExit-Isolated, $(G; \overrightarrow{L}; \texttt{exit})$ done where:

(b) If there are other processes running:

i. By case assumption for TExit, P has form `exit` and there exists another process that is running. As such, OPExit-Continue applies.

ii. By OPExit-Continue, $(G; \overrightarrow{L}; (P_1, \ldots, \texttt{exit}, \ldots, P_n)) \mapsto (G; \overrightarrow{L}; (P_1, \ldots, P_n))$ where:

Case 14: (TExec-Prog) $\dfrac{\Gamma; \Delta, c : \lceil \neg \rho \rceil \vdash Q \quad \cdot; n : \lceil \rho \rceil \vdash P}{\Gamma; \Delta \vdash (c = \texttt{exec}\ \ P :: n : \rho).Q}$

(a) By case assumption for TExec-Prog, P has form $(c = \texttt{exec}\ \ Q :: n : \rho).R$. As such, OPExec-Prog applies.

(b) By OPExec-Prog, $(G; L_i; (c = \texttt{exec}\ \ P :: n : \rho).Q) \mapsto (G'; (L_i', L_P'); (P, Q))$ where:

- $G \equiv \overrightarrow{z \mapsto q}$
- $G' \equiv \overrightarrow{z \mapsto q}, fn_c \mapsto [], fn_n \mapsto []$
- $L_i' \equiv L_i[c \mapsto fn_c]$
- $L_P' \equiv \{n \mapsto fn_n\}$
- $\overline{c} = n$
- $\overline{n} = c$

(c) Thus, by OPUniversalStep, the entire system steps.

□

H Preservation Proof

Proof. If $(G; \overrightarrow{L}; \overrightarrow{P})$ ok and $(G; \overrightarrow{L}; \overrightarrow{P}) \mapsto (G'\overrightarrow{L'}; \overrightarrow{P'})$ then $(G'\overrightarrow{L'}; \overrightarrow{P'})$ ok.

Case 1: (OPExit-Continue) $(G; \overrightarrow{L}; (P_1, \ldots, \texttt{exit}, \ldots, P_n)) \mapsto (G; \overrightarrow{L}; (P_1, \ldots, P_n))$

(a) By TypeL and OK, typechecking each process occurs independently of typechecking all other processes.

(b) As this operation only removes P_i from the configuration, typechecking $(G; \overrightarrow{L}; \overrightarrow{P'})$ is the same as typechecking $(G; \overrightarrow{L}; \overrightarrow{P})$ with the exception of typechecking P_i.

(c) By our IH, $(G; \overrightarrow{L}; \overrightarrow{P})$ ok.

(d) Thus, $(G; \overrightarrow{L}; \overrightarrow{P'})$ ok as the rest of the system already typechecks without P_i.

Case 2: (OPUniversalStep) $\dfrac{(G; L_i; P_i) \mapsto (G'; L_i'; P_i')}{(G; \overrightarrow{L}; \overrightarrow{P}) \mapsto (G'; \overrightarrow{L'}; \overrightarrow{P'})}$

Subcase 1: (OPSend) $(G; L_i; c\langle e\rangle.P) \mapsto (G'; L_i\{e\}; P)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash c\langle e\rangle$

ii. And, by $(G; L_i; c\langle e\rangle.P) \mapsto (G'; L_i\{e\}; P)$ (where τ is the type of the sent expression e):
- $(G; L_i) \vdash e \Downarrow v$
- $G \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overrightarrow{c} \mapsto \overrightarrow{m}, v$

iii. Where:
- $L_i' \equiv L_i\{e\}$
- $P_i' \equiv P$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

v. Let $\Gamma_i' \equiv \Gamma$

vi. Let $\Delta_i' \equiv \Delta\{e\}$

vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G[\overline{c} \mapsto \Sigma_B^G(\overline{c}); v[\tau]]$

viii. Let $\Sigma_O \equiv \Sigma_O'[\overline{c} \mapsto +\tau; \Sigma_O'(\overline{c})]$

ix. Then:

A. $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

I. By TypeL, need to show $\Delta_i'(\overline{c}) = \Sigma_B^{G'}(\Sigma_O')(\overline{c}) = \Delta_i(\overline{c}) = \Sigma_B^G(\Sigma_O)(\overline{c})$.

II. By definition of $\Sigma_B^G(\rho)$,
$\Sigma_B^{\overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, v[\tau]}(\Sigma_O')(\overline{c}) = \Sigma_B^G(\overline{c}); +\tau; \Sigma_O'(\overline{c}) = \Sigma_B^G(\overline{c}); \Sigma_O(\overline{c})$

B. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and TSend)

x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P_i'})$ ok by OK.

Subcase 2: (OPRecv) $(G; L_i; c(x).P) \mapsto (G'; L_i[x \mapsto v]; P)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash c(x)$
ii. And, by $(G; L_i; c(x).P) \mapsto (G'; L_i[x \mapsto v]; P)$ (where τ is the type of received x):
- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto v, \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Where:
- $L_i' \equiv L_i[x \mapsto v]$
- $P_i' \equiv P$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$
v. Let $\Gamma_i' \equiv \Gamma\langle x \rangle$
vi. Let $\Delta_i' \equiv \Delta\langle x \rangle$
vii. Let $\Sigma_O' \equiv \Sigma_O$
viii. Let $\Sigma_B^G \equiv \Sigma_B^G[c \mapsto x[\tau]; \Sigma_B^{G'}(c)]$
ix. Then:
A. $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$
I. By IH, we know $\Delta_i(c) = \Sigma_B^G(\Sigma_O)(c) = +\tau; \rho$
II. By TypeL, need to show $\Delta_i'(c) = \Sigma_B^{G'}(\Sigma_O')(c) = \rho$
III. By definition of $\Sigma_B^G(\rho)$,
$\Sigma_B^G(\Sigma_O)(c) = +\tau; \Sigma_B^{\overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}}(\Sigma_O)(c) = +\tau; \Sigma_B^{G'}(\Sigma_O')(c) = +\tau; \rho$
IV. Thus, $\Delta_i'(x) = \Sigma_B^{G'}(\Sigma_O')(c) = \rho$.
B. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and TRecv)
x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 3: (OPCase1) $(G; L_i; c.\mathsf{case}(P,\ Q).R) \mapsto (G'; L_i; P.R)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash c.\mathsf{case}(P,\ Q).R$
ii. And, by $(G; L_i; c.\mathsf{case}(P,\ Q).R) \mapsto (G'; L_i; P.R)$ (where ρ_1 is the protocol recognized by $P.R$ and ρ_2 is the protocol recognized by $Q.R$):
- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{SEL}[\cdot\ \&\ \rho_2], \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Where:
- $L_i' \equiv L_i$
- $P_i' \equiv P.R$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$
v. Let $\Gamma_i' \equiv \Gamma$
vi. Let $\Delta_i' \equiv \Delta, c : \lceil \rho_1 \rceil$

vii. Let $\Sigma_B^G \equiv \Sigma_B^G[c \mapsto \mathsf{SEL}[\cdot \,\&\, \rho_2]; \Sigma_B^{G'}(c)]$
viii. Let $\Sigma'_O \equiv \Sigma_O$
ix. Then:
A. $\Sigma_B^{G'}(\Sigma'_O) \vdash L'_i : (\Gamma'_i; \Delta'_i)$
I. By TypeL, need to show $\Delta'_i(c) = \Sigma_B^{G'}(\Sigma'_O)(c) = \rho_1$.
II. By IH, $\Delta_i(c) = \Sigma_B^G(\Sigma_O)(c) = \rho_1 \,\&\, \rho_2$
III. By definition of $\Sigma_B^G(\rho)$, $\Sigma_B^G(\Sigma_O)(c) = \Sigma_B^{G'}(\Sigma'_O)(c) \,\&\, \rho_2$
IV. Thus, $\Delta'_i(c) = \Sigma_B^{G'}(\Sigma'_O)(c) = \rho_1$.
B. $(\Gamma'_i; \Delta'_i) \vdash P'_i$ (from D2 and TCase)
x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 4: (OPCase2) $(G; L_i; c.\mathtt{case}(P,\ Q).R) \mapsto (G'; L_i; Q.R)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash c.\mathtt{case}(P,\ Q).R$
ii. And, by $(G; L_i; c.\mathtt{case}(P,\ Q).R) \mapsto (G'; L_i; Q.R)$ (where ρ_1 is the protocol recognized by $P.R$ and ρ_2 is the protocol recognized by $Q.R$):
• $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{SEL}[\rho_1 \,\&\, \cdot], \overrightarrow{m}$
• $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$
iii. Where:
• $L'_i \equiv L_i$
• $P'_i \equiv Q.R$
iv. Want to show that $(G'; L'_i; P'_i)$ ok. To do this, we need to pick $\Sigma'_B; \Sigma'_O; \Gamma'_i; \Delta'_i$
v. Let $\Gamma'_i \equiv \Gamma$
vi. Let $\Delta'_i \equiv \Delta, c : \restriction \rho_1 \restriction$
vii. Let $\Sigma_B^G \equiv \Sigma_B^G[c \mapsto \mathsf{SEL}[\rho_1 \,\&\, \cdot]; \Sigma_B^{G'}(c)]$
viii. Then:
A. $\Sigma_B^{G'}(\Sigma'_O) \vdash L'_i : (\Gamma'_i; \Delta'_i)$
I. By IH, $\Delta_i(c) = \Sigma_B^G(\Sigma_O)(c) = \rho_1 \,\&\, \rho_2$
II. By TypeL, need to show $\Delta'_i(c) = \Sigma_B^{G'}(\Sigma'_O)(c) = \rho_2$.
III. By definition of $\Sigma_B^G(\rho)$, $\Sigma_B^G(\Sigma_O)(c) = \rho_1 \,\&\, \Sigma_B^{G'}(\Sigma'_O)(c)$
IV. Thus, $\Delta'_i(c) = \Sigma_B^{G'}(\Sigma'_O)(c) = \rho_2$.
B. $(\Gamma'_i; \Delta'_i) \vdash P'_i$ (from D2 and TCase)
ix. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 5: (OPSelect1) $(G; L_i; c[1].P) \mapsto (G'; L_i; P)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash c[1].P$

ii. And, by $(G; L_i; c[1].P) \mapsto (G'; L_i; P)$:
- $G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[\cdot \,\&\, \rho_2]$

iii. Where:
- $L_i' \equiv L_i$
- $P_i' \equiv P$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

v. Let $\Gamma_i' \equiv \Gamma$

vi. Let $\Delta_i' \equiv \Delta, c : \lceil \rho_1 \rceil$

vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G[\overline{c} \mapsto \Sigma_B^G(\overline{c}); \mathsf{SEL}[\cdot \,\&\, \rho_2]]$

viii. Let $\Sigma_O' \equiv \Sigma_O[\overline{c} \mapsto \rho_1]$

ix. Then:

A. $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

I. By TypeL, need to show $\Delta_i'(\overline{c}) = \Sigma_B^{G'}(\Sigma_O')(\overline{c}) = \Delta_i(\overline{c}) = \Sigma_B^G(\Sigma_O)(\overline{c})$

II. By definition of $\Sigma_B^G(\rho)$, $\Sigma_B^{\overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[\cdot \,\&\, \rho_2]}(\Sigma_O')(\overline{c}) = \Sigma_B^G(\overline{c}); \Sigma_O'(\overline{c}) \,\&\, \rho_2 = \Sigma_B^G(\Sigma_O)(\overline{c})$

B. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and TSelect1)

x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 6: (OPSelect2) $(G; L_i; c[2].P) \mapsto (G'; L_i; P)$

i. From OK, we have the premises:

D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$

D2: $(\Gamma_i; \Delta_i) \vdash c[2].P$

ii. And, by $(G; L_i; c[2].P) \mapsto (G'; L_i; P)$:
- $G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[\rho_1 \,\&\, \cdot]$

iii. Where:
- $L_i' \equiv L_i$
- $P_i' \equiv P$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

v. Let $\Gamma_i' \equiv \Gamma$

vi. Let $\Delta_i' \equiv \Delta, c : \lceil \rho_1 \rceil$

vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G[\overline{c} \mapsto \Sigma_B^G(\overline{c}); \mathsf{SEL}[\rho_1 \,\&\, \cdot]]$

viii. Let $\Sigma_O \equiv \Sigma_O'[\overline{c} \mapsto \rho_1 \,\&\, \rho_2; \Sigma_O'(\overline{c})]$

ix. Then:

A. $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

I. By TypeL, need to show $\Delta_i'(\overline{c}) = \Sigma_B^{G'}(\Sigma_O')(\overline{c}) = \Delta_i(\overline{c}) = \Sigma_B^G(\Sigma_O)(\overline{c})$.

II. $\Sigma_B^{\overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{SEL}[\rho_1 \,\&\, \cdot]}(\Sigma_O')(\overline{c}) = \Sigma_B^G(\overline{c}); \rho_1 \,\&\, \Sigma_O'(\overline{c}) = \Sigma_B^G(\Sigma_O)(\overline{c})$

B. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and TSelect2)

x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 7: (OPWhileT) $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; P.\texttt{while}(e)\{P\}.Q)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{while}(e)\{P\}.Q$

ii. And, by $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; P.\texttt{while}(e)\{P\}.Q)$:
- $(G; L_i) \vdash e \Downarrow \mathsf{true}$

iii. Where:
- $G' \equiv G$
- $L_i' \equiv L_i$
- $P_i \equiv \texttt{while}(e)\{P\}.Q$
- $P_i' \equiv P.P_i$

iv. Want to show that $(G'; L_i'; P_i')$ ok.

v. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

vi. Let $\Gamma_i' \equiv \Gamma$

vii. Let $\Delta_i' \equiv \quad \left(\Delta_1 \equiv \{x_i : \restriction^\bullet \rho_i \restriction \text{ for all } x_i \in \Delta\}\right)$

viii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G$

ix. Let $\Sigma_O' \equiv \Sigma_O$

x. In addition, we need to show:
$(G'; L_i'; P.\texttt{while}(e)\{P\}.Q) \mapsto (G''; L_i''; \texttt{while}(e)\{P\}.Q)$ ok

xi. To do this, we need to pick $\Sigma_B^{G''}; \Sigma_O''; \Gamma_i''; \Delta_i''$

xii. Let $\Gamma_i'' \equiv \Gamma_i$

xiii. Let $\Delta_i'' \equiv \Delta_i$

xiv. Let $\Sigma_B^{G''} \equiv \Sigma_B^{G'}$

xv. Let $\Sigma_O'' \equiv \Sigma_O$

xvi. Then:

Part i: $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

I. Because $\Gamma; \Delta_1 \vdash P$ and $\Gamma; \Delta \vdash P_i$, the Guarded Step Lemma applies: $\Gamma; \Delta \vdash P.P_i \equiv \Gamma; \Delta \vdash P_i'$.

II. Thus, $\Sigma_B^G(\Sigma_O) = \Sigma_B^{G'}(\Sigma_O') = \Sigma_B^{G''}(\Sigma_O'')$.

III. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from the Guarded Step Lemma and TWhile)

Part ii: $\Sigma_B^{G''}(\Sigma_O'') \vdash L_i'' : (\Gamma_i''; \Delta_i'') \equiv \Sigma_B^{G'}(\Sigma_O) \vdash L_i'' : (\Gamma_i; \Delta_i)$

I. By IH, $L_i'' \equiv L_i$. Thus, $\Sigma_B^{G''}(\Sigma_O'') \vdash L_i'' : (\Gamma_i''; \Delta_i'') \equiv \Sigma_B^{G'}(\Sigma_O') \vdash L_i : (\Gamma_i; \Delta_i)$

II. By TypeL, need to show that $\Sigma_B^{G''}(\Sigma_O'') \equiv \Sigma_B^{G'}(\Sigma_O) = \Sigma_B^G(\Sigma_O)$ which was proven in *Part i*.

III. $(\Gamma_i''; \Delta_i'') \vdash P_i'' \equiv (\Gamma_i; \Delta_i) \vdash P_i$ (from the Guarded Step Lemma, TWhile, and IH)

xvii. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 8: (OPWhileF) $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; Q)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{while}(e)\{P\}.Q$
ii. And, by $(L_i; \texttt{while}(e)\{P\}.Q) \mapsto (L_i; Q)$:
• $(G; L_i) \vdash e \Downarrow \mathsf{false}$
iii. Where:
• $G' \equiv G$
• $L_i' \equiv L_i$
• $P_i' \equiv Q$
iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$
v. Let $\Gamma_i' \equiv \Gamma$
vi. Let $\Delta_i' \equiv \Delta$
vii. Let $\Sigma_B' \equiv \Sigma_B$
viii. Let $\Sigma_O' \equiv \Sigma_O$
ix. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from IH, D2 and TWhile)
x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 9: (OP?Unfold) $(G; L_i; \texttt{more}(c).P) \mapsto (G'; L_i; P)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{more}(c).P$
ii. And, by $(G; L_i; \texttt{more}(c).P) \mapsto (G'; L_i; P)$:
• $G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}$
• $G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{START{-}LOOP}$
iii. Where:
• $L_i' \equiv L_i$
• $P_i' \equiv P$
iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$
v. Let $\Gamma_i' \equiv \Gamma$
vi. Let $\Delta_i' \equiv \Delta, c : \lceil \rho_1; ?\rho_1; \rho_2 \rceil$ or $\Delta_i' \equiv \Delta, c : \lceil \rho_1; {}^\bullet?\rho_1; \rho_2 \rceil$
vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G[\overline{c} \mapsto \Sigma_B^G(\overline{c}); \mathsf{START{-}LOOP}]$
viii. Let $\Sigma_O' \equiv \Sigma_O[\overline{c} \mapsto \neg\rho_1; \Sigma_O(\overline{c})]$ where $\Sigma_O(\overline{c})$ of form $!\neg\rho_1; \rho_2$ (as $\overline{c}$ is unaware of if c is guarded or not)
ix. Then:
A. $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$
I. By TypeL, need to show $\Delta_i'(\overline{c}) = \Sigma_B^{G'}(\Sigma_O')(\overline{c}) = \Delta_i(\overline{c}) = \Sigma_B^G(\Sigma_O)(\overline{c})$.
II. By definition of $\Sigma_B^G(\rho)$, $\Sigma_B^{\overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{START{-}LOOP}}(\Sigma_O')(\overline{c}) = \Sigma_B^G(\overline{c}); (\mathsf{let}\ \alpha; !\alpha; \beta = \Sigma_O'(\overline{c})\ \mathsf{in}\ !\alpha; \beta) = \Sigma_B^G(\overline{c}); \Sigma_O(\overline{c})$
B. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and T?Unfold or D2 and T?Unfold Guarded)

x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 10: (OP?Weaken) $(G; L_i; \texttt{weaken}(c).P) \mapsto (G'; L_i; P)$

i. From OK, we have the premises:

D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$

D2: $(\Gamma_i; \Delta_i) \vdash \texttt{weaken}(c).P$

ii. And, by $(G; L_i; \texttt{weaken}(c).P) \mapsto (G'; L_i; P)$:

- $G \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{END{-}LOOP}[!\neg\rho_1]$

iii. Where:

- $L_i' \equiv L_i$
- $P_i' \equiv P$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

v. Let $\Gamma_i' \equiv \Gamma$

vi. Let $\Delta_i' \equiv \Delta, c : \uparrow ?\rho_1; \rho_2 \uparrow$

vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G[\overline{c} \mapsto \Sigma_B^G(\overline{c}); \mathsf{END{-}LOOP}[!\neg\rho_1]]$

viii. Let $\Sigma_O \equiv \Sigma_O'[\overline{c} \mapsto !\neg\rho_1; \Sigma_O'(\overline{c})]$

ix. Then:

A. $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

I. By TypeL, need to show $\Delta_i'(\overline{c}) = \Sigma_B^{G'}(\Sigma_O')(\overline{c}) = \Delta_i(\overline{c}) = \Sigma_B^G(\Sigma_O)(\overline{c})$.

II. By definition, $\Sigma_B^{\overrightarrow{z \mapsto q}, \overline{c} \mapsto \overrightarrow{m}, \mathsf{END{-}LOOP}[!\neg\rho_1]}(\Sigma_O')(\overline{c}) = \Sigma_B^G(\overline{c}); !\neg\rho_1; \Sigma_O'(\overline{c}) = \Sigma_B^G(\overline{c}); \Sigma_O(\overline{c})$

B. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and T?Weaken)

x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 11: (OPAccept-Loop) $(G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{accept}(c)\{P\}.Q)$

i. From OK, we have the premises:

D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$

D2: $(\Gamma_i; \Delta_i) \vdash \texttt{accept}(c)\{P\}.Q$

ii. And, by $(G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; P.\texttt{accept}(c)\{P\}.Q)$:

- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Where:

- $L_i' \equiv L_i$
- $P_i \equiv \texttt{accept}(c)\{P\}.Q$
- $P_i' \equiv P.P_i$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B^{G'}; \Sigma_O'; \Gamma_i'; \Delta_i'$

v. Let $\Gamma_i' \equiv \Gamma$

vi. Let $\Delta_i' \equiv \quad \big(\Delta_1 \equiv \{x_i : \uparrow^\bullet \rho_i \uparrow \text{ for all } x_i \in \Delta\}\big) \;, c : \uparrow \rho_1 \uparrow$

vii. Let $\Sigma_B^G \equiv \Sigma_B^G[c \mapsto \mathsf{START{-}LOOP}; \Sigma_B^{G'}(c)]$

viii. Let $\Sigma_O \equiv \Sigma'_O[c \mapsto \rho_1; \Sigma'_O(c)]$
ix. In addition, we need to show that $(G'; L'_i; P'_i) \mapsto (G''; L''_i; P_i)$ ok
x. To do this, we need to pick $\Sigma_B^{G''}; \Sigma''_O; \Gamma''_i; \Delta''_i$
xi. Let $\Gamma''_i \equiv \Gamma_i$
xii. Let $\Delta''_i \equiv \Delta_i$
xiii. Let $\Sigma_B^{G''} \equiv \Sigma_B^{G'}$
xiv. Let $\Sigma''_O \equiv \Sigma_O$
xv. Then:

Part i: $\Sigma_B^{G'}(\Sigma'_O) \vdash L'_i : (\Gamma'_i; \Delta'_i)$

I. Because $\Gamma; \Delta_1 \vdash P$ and $\Gamma; \Delta \vdash P_i$, the Guarded Step Lemma applies: $\Gamma; \Delta \vdash P.P_i$.
II. Thus, $\Sigma_B^G(\Sigma_O) \equiv \Sigma_B^{G'}(\Sigma'_O) \equiv \Sigma_B^{G''}(\Sigma''_O)$.
III. $(\Gamma'_i; \Delta'_i) \vdash P'_i$ (from the Guarded Step Lemma and TAccept)

Part ii: $\Sigma_B^{G''}(\Sigma''_O) \vdash L''_i : (\Gamma''_i; \Delta''_i)$

I. By IH, $L''_i \equiv L_i$. Thus, $\Sigma_B^{G''}(\Sigma''_O) \vdash L''_i : (\Gamma''_i; \Delta''_i) = \Sigma_B^{G'}(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
II. By TypeL, need to show that $\Sigma_B^{G''}(\Sigma''_O) \equiv \Sigma_B^{G'}(\Sigma'_O) = \Sigma_B^G(\Sigma_O)$ which was proven in *Part i*.

xvi. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 12: (OPAccept-End) $(G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; Q)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{accept}(c)\{P\}.Q$
ii. And, by $(G; L_i; \texttt{accept}(c)\{P\}.Q) \mapsto (G'; L_i; Q)$:
- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Where:
- $L'_i \equiv L_i$
- $P'_i \equiv Q$

iv. Want to show that $(G'; L'_i; P'_i)$ ok. To do this, we need to pick $\Sigma'_B; \Sigma'_O; \Gamma'_i; \Delta'_i$
v. Let $\Gamma'_i \equiv \Gamma$
vi. Let $\Delta'_i \equiv \Delta$
vii. Let $\Sigma_B^G \equiv \Sigma_B^G[c \mapsto \mathsf{END{-}LOOP}[!\rho_1]; \Sigma_B^{G'}(c)]$
viii. Let $\Sigma_O \equiv \Sigma_O[c \mapsto !\rho_1; \Sigma'_O(c)]$
ix. Then:
A. $\Sigma_B^{G'}(\Sigma'_O) \vdash L'_i : (\Gamma'_i; \Delta'_i)$
I. By IH, we know $\Delta_i(c) = \Sigma_B^G(\Sigma_O)(c) = !\rho_1; \rho_2$.
II. By TypeL, need to show $\Delta'_i(c) = \Sigma_B^{G'}(\Sigma'_O)(c) = \rho_2$.
III. By definition of $\Sigma_B^G(\rho)$, $\Sigma_B^G(\Sigma_O)(c) = !\rho_1; \Sigma_B^{G'}(\Sigma'_O)(c)$
B. $(\Gamma'_i; \Delta'_i) \vdash P'_i$ (from D2 and TAccept)

x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P_i'})$ ok by OK.

Subcase 13: (OPAcceptWhile-Loop) $\begin{array}{l}(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto \\ (G'; L_i; P.\texttt{acceptWhile}(c,\ e)\{P\}.Q)\end{array}$

i. From OK, we have the premises:

D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$

D2: $(\Gamma_i; \Delta_i) \vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q$

ii. And, by $\begin{array}{l}(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto \\ (G'; L_i; P.\texttt{acceptWhile}(c,\ e)\{P\}.Q)\end{array}$:

- $(G; L_i) \vdash e \Downarrow \mathsf{true}$
- $G \equiv \overrightarrow{z \mapsto q}, c \mapsto \mathsf{START{-}LOOP}, \overrightarrow{m}$
- $G' \equiv \overrightarrow{z \mapsto q}, c \mapsto \overrightarrow{m}$

iii. Where:

- $L_i' \equiv L_i$
- $P_i \equiv \texttt{acceptWhile}(c,\ e)\{P\}.Q$
- $P_i' \equiv P.P_i$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B^{G'}; \Sigma_O'; \Gamma_i'; \Delta_i'$

v. Let $\Gamma_i' \equiv \Gamma$

vi. Let $\Delta_i' \equiv \quad \left(\Delta_1 \equiv \{x_i : \lceil \bullet \rho_i \rceil \text{ for all } x_i \in \Delta\}\right)\ , c : \lceil \rho_1 \rceil$

vii. Let $\Sigma_B^G \equiv \Sigma_B^G[c \mapsto \mathsf{START{-}LOOP}; \Sigma_B^{G'}(c)]$

viii. Let $\Sigma_O \equiv \Sigma_O'[c \mapsto \rho_1; \Sigma_O'(c)]$

ix. In addition, we need to show that $(G'; L_i'; P_i') \mapsto (G''; L_i''; P_i)$ ok

x. To do this, we need to pick $\Sigma_B^{G''}; \Sigma_O''; \Gamma_i''; \Delta_i''$

xi. Let $\Gamma_i'' \equiv \Gamma_i$

xii. Let $\Delta_i'' \equiv \Delta_i$

xiii. Let $\Sigma_B^{G''} \equiv \Sigma_B^{G'}$

xiv. Let $\Sigma_O'' \equiv \Sigma_O$

xv. Then:

Part i: $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

I. Because $\Gamma; \Delta_1 \vdash P$ and $\Gamma; \Delta \vdash P_i$, the Guarded Step Lemma applies: $\Gamma; \Delta \vdash P.P_i$.

II. Thus, $\Sigma_B^G(\Sigma_O) \equiv \Sigma_B^{G'}(\Sigma_O') \equiv \Sigma_B^{G''}(\Sigma_O'')$.

III. $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from the Guarded Step Lemma and TAcceptWhile or TAcceptWhile Guarded)

Part ii: $\Sigma_B^{G''}(\Sigma_O'') \vdash L_i'' : (\Gamma_i''; \Delta_i'')$

I. By IH, $L_i'' \equiv L_i$. Thus, $\Sigma_B^{G''}(\Sigma_O'') \vdash L_i'' : (\Gamma_i''; \Delta_i'') = \Sigma_B^{G'}(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$

II. By TypeL, need to show that $\Sigma_B^{G''}(\Sigma_O'') \equiv \Sigma_B^{G'}(\Sigma_O) = \Sigma_B^G(\Sigma_O)$ which was proven in *Part i*.

xvi. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'})$ ok by OK.

Subcase 14: (OPAcceptWhile-Preempt) $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q$
ii. And, by $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)$:
- $(G; L_i) \vdash e \Downarrow \mathsf{false}$
- $G' \equiv G$

iii. Where:
- $L_i' \equiv L_i$
- $P_i' \equiv Q$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$
v. Let $\Gamma_i' \equiv \Gamma$
vi. Let $\Delta_i' \equiv \Delta$
vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G$
viii. Let $\Sigma_O' \equiv \Sigma_O$
ix. Then, $\Sigma_B^{G'}(\Sigma_O') \equiv \Sigma_B^G(\Sigma_O)$, which is our IH.
x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P_i'})$ ok by OK.

Subcase 15: (OPAcceptWhile-End) $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{acceptWhile}(c,\ e)\{P\}.Q$
ii. And, by $(G; L_i; \texttt{acceptWhile}(c,\ e)\{P\}.Q) \mapsto (G'; L_i; Q)$:
- $G \equiv z \mapsto \overrightarrow{q}, c \mapsto \mathsf{END{-}LOOP}[!\rho_1], \overrightarrow{m}$
- $G' \equiv G$

iii. Where:
- $L_i' \equiv L_i$
- $P_i' \equiv Q$

iv. Want to show that $(G'; L_i'; P_i')$ ok. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$
v. Let $\Gamma_i' \equiv \Gamma$
vi. Let $\Delta_i' \equiv \Delta$
vii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G$
viii. Let $\Sigma_O' \equiv \Sigma_O$
ix. Then, $\Sigma_B^{G'}(\Sigma_O') \equiv \Sigma_B^G(\Sigma_O)$, which is our IH.
x. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P_i'})$ ok by OK.

Subcase 16: (OPClose) $(L_i; \texttt{close}(c).P) \mapsto (L_i; P)$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{close}(c).P$
ii. And, by $(L_i; \texttt{close}(c).P) \mapsto (L_i; P)$:
- $G' \equiv G$
- $L_i' \equiv L_i$

iii. Thus, $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and TClose)

Subcase 17: (OPSkip-Exit) $(G; L_i; \texttt{skip}) \mapsto (G; L_i; \texttt{exit})$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash \texttt{skip}$

ii. And, by $(G; L_i; \texttt{skip}) \mapsto (G; L_i; \texttt{exit})$

iii. By (TSkip) $\dfrac{}{\Gamma; \Delta \vdash \texttt{skip}}$ $\left(\text{For all } x \in \Delta, \Delta(x) = \uparrow^\bullet \rho \uparrow\right)$, Δ_i must be comprised entirely of guarded resources.

iv. Thus, the Guarded Step Lemma applies where P is `skip` and Q is the empty process.

v. By the Guarded Step Lemma, P must preserve all guarded resources for Q to utilize.

vi. Because Q (as the empty process) uses no resources and $(\Gamma_i; \Delta_i) \vdash \texttt{skip}$ typechecks per our IH, then Δ_i must be empty.

vii. Thus, $(\Gamma_i; \Delta_i) \vdash \texttt{exit}$.

Subcase 18: (OPExec-Prog) $(G; L_i; (c = \texttt{exec}\ \ P :: n : \rho).Q) \mapsto (G'; (L_i', L_P'); (P, Q))$

i. From OK, we have the premises:
D1: $\Sigma_B^G(\Sigma_O) \vdash L_i : (\Gamma_i; \Delta_i)$
D2: $(\Gamma_i; \Delta_i) \vdash c\langle e \rangle$

ii. And, by $(G; L_i; (c = \texttt{exec}\ \ P :: n : \rho).Q) \mapsto (G'; (L_i', L_P'); (P, Q))$:
- $G \equiv \overrightarrow{z \mapsto q}$
- $G' \equiv \overrightarrow{z \mapsto q}, fn_c \mapsto [], fn_n \mapsto []$
- $L_i' \equiv L_i[c \mapsto fn_c]$
- $L_P' \equiv \{n \mapsto fn_n\}$
- $\overline{c} = n$
- $\overline{n} = c$

iii. Let $\Sigma_B^{G'} \equiv \Sigma_B^G[fn_c \mapsto [], fn_n \mapsto []]$

iv. Let $\Sigma_O' \equiv \Sigma_O[c \mapsto \neg\rho, n \mapsto \rho]$

Case 1: $P_i' \equiv Q$

I. Want to show that $(G'; L_i'; P_i')$ `ok`. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

II. Let $\Gamma_i' \equiv \cdot$

III. Let $\Delta_i' \equiv n : \uparrow\rho\uparrow$

IV. Then:

(1) $\Sigma_B^{G'}(\Sigma_O') \vdash L_i' : (\Gamma_i'; \Delta_i')$

(I) By TypeL, need to show $\Delta_i'(x) = \Sigma_B^{G'}(\Sigma_O')(fn_n)$.

(II) $\Sigma_B^{\overrightarrow{z \mapsto q}, fn_n \mapsto []}(\Sigma_O')(fn_n)$ then, $\Delta_i'(fn_n) = \Sigma_O' = \rho$

(2) $(\Gamma_i'; \Delta_i') \vdash P_i'$ (from D2 and TExec-Prog)

Case 2: $P_i' \equiv P$

I. Want to show that $(G'; L_i'; P_i')$ `ok`. To do this, we need to pick $\Sigma_B'; \Sigma_O'; \Gamma_i'; \Delta_i'$

II. Let $\Gamma_i' \equiv \Gamma$

III. Let $\Delta'_i \equiv \Delta, c : \lceil \neg \rho \rceil$

IV. Then:

(1) $\Sigma_B^{G'}(\Sigma'_O) \vdash L'_i : (\Gamma'_i; \Delta'_i)$

(I) By TypeL, need to show $\Delta'_i(x) = \Sigma_B^{G'}(\Sigma'_O)(fn_c)$.

(II) $\Sigma_B^{\overrightarrow{z \mapsto q}, fn_c \mapsto []}(\Sigma'_O)(fn_c)$ then, $\Delta'_i(fn_c) = \Sigma'_O = \rho$

(2) $(\Gamma'_i; \Delta'_i) \vdash P'_i$ (from D2 and TExec-Prog)

v. Thus, we can conclude that $(G'; \overrightarrow{L'}; \overrightarrow{P'_i})$ ok by OK.

□