The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering, Computing and Cybernetics (CECC)

Using Redex for Safe Programming Languages Research: A Case Study in System C

— 24 pt research project (S1/S2 2024)

A report submitted for the course
COMP4550, Computing Research Project

By:
Ziling Ouyang

Supervisors:
Dr. Alex Potanin
Dr. Fabian Muehlboeck

June 2025

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

June, Ziling Ouyang

Abstract

Proofs in programming languages research are the established way to formalise the safety of a language or proposed new features for a language. As such, they are a crucial part of most research which aims to introduce new programming languages. However, proofs which are done by hand can be rather extensive and, thus, prone to human error. For this reason, mechanical provers such as Coq are sometimes used to augment the proof of language safety to ensure correctness for a proof. Although this addition of mechanised provers does verify proof correctness, it also tends to be even more complex than just writing a proof by hand. Therefore, we attempt to introduce an approach which uses PLT Redex as an addition to non-mechanised safety proofs. This proposed approach aims to provide some of the safeguards associated with mechanised provers without all of their complexity. We further demonstrate the effectiveness of this approach by performing two case studies which extend the System C language while using the novel approach.

Table of Contents

Chapter 1

Introduction

The concept of safety or soundness has long been a fundamental part of research in the programming languages field. A safety proof for a programming language is an indication that well-formed programs in the language do not exhibit dangerous and unintended behaviour. Or in other words, "well-type[d] programs cannot 'go wrong'" (Milner, 1978) and a safe programming language only accepts well-typed programs. Furthermore, safety proofs have the ability of being able to detect errors in existing languages which may not have been formally proved. An example of this is Syme (1999)'s paper proving the safety of the Java language wherein errors were discovered in Java due during the proof.

In recent years, the notion of safe programming languages has been gaining popularity. There has been research into type safe variants of the C programming language (Li, 2004) such as Cyclone (Jim et al., 2002) and Vault (Fahndrich and DeLine, 2002). Additionally, attempts have been made into automatically translating programs from unsafe languages into those written in safe languages. A recent example of this is Emre et al. (2021) where the authors of the paper investigate translating C programs into Rust programs so that potential security vulnerabilities associated with unsafety can be eliminated. Moreover, a plethora of modern type safe languages have been gaining traction. Wyvern (Melicher et al., 2022), Scala (Odersky et al., 2004) and Rust (Klabnik and Nichols, 2023) all exist in this category of type safe language.

While it may seem obvious, all safe languages must have been proved safe before they can be guaranteed to not exhibit dangerous and unintended behaviour. However, this proof of safety is often very repetitive, if not outright tedious, and prone to human error from the researcher. As such, in order to efficiently present a formal proof of safety for novel extensions to languages or entire languages, we present an approach which attempts to make the process of doing research on type safe languages more productive. This approach utilises PLT Redex (Flatt and PLT, 2010) to model a language which is being worked on. As a result, more intuition can be gained for the language and obvious errors can be detected before a proof of safety is even considered. In addition, this approach also allows for a faster iteration cycle due to a living, breathing model of the language being present for querying instead of the researcher having to rely on continual proofs of safety. We evaluate this approach through case studies performed in System C. In Chapter 8, we discuss the case studies, approach, and whether the approach helped with developing a proof of safety for the extensions to System C presented in this thesis.

1.1 Motivation

This thesis was motivated by prior work which was completed by the author (Ouyang, 2023) and also a sibling thesis by Ou (2023). Both of these papers attempted to combine The Wyvern Developers (2018)'s Wyvern language with the System C language in a way which would explore the interplay of the effect systems in the two languages when integrated. The fundamental direction of research was to implement and prove the safety of the design introduced within the Ouyang (2023) paper while referring to Ou (2023) as a basis for the safety proof. However, as the proof of safety was left partially incomplete in Ou (2023) due to complications with the variable store, the direction for this research instead pivoted to adding additional extensions to the System C language while completing a full formal proof of safety.

However, it was discovered that a formal proof of safety for any language can be a tedious process and especially so when multiple extensions to a language are being considered. There often arose complications such as those in Ou (2023) or even human errors due to how long the proof was. This flaw in manually doing a proof of safety becomes more obvious when it is considered in the larger context of language research. There is no good place to attempt a proof of safety when multiple features are being added to a type safe language. If the proof of safety is only done at the end, this may result in inefficiencies if the safety proof detects errors in the language features. On the other hand, continuously writing and updating a proof of safety is time expensive and wasteful. Prompted by this dilemma, we came up with our approach which uses Redex to address some of the issues associated with proving safety in language research.

1.2 Contributions

This thesis has the following contributions:

- Developed an approach for proving language features in an iterative manner.
- Performed a case study using the proposed approach to implement variables into System C.
- Performed an additional case study with the proposed approach on extending System C with simple handler abstraction.
- Evaluated the approach based on the outcomes of the case studies.

- Extended the System C language with variables and simple handler abstraction using the approach introduced.
- Provided a model of the System C language in Redex to improve intuition around language features and mechanisms. This can be found at `https://github.com/Purling/system_c_racket`.

Finally, there is a discussion of limitations, lines of future work and conclusion in Chapter 8 and Chapter 9.

Chapter 2

Background and Related Work

We start off by introducing some of the concepts which are key to understanding the ideas presented in this paper. This chapter will begin by providing an overview of Redex before exploring effect and capabilities - two core concepts behind System C. To conclude, we will briefly examine System C and language safety.

2.1 PLT Redex

Programming Language Team (PLT) Redex, or just Redex, is a library embedded into the Racket language which allows for the specification and modeling of programming languages (Findler et al., 2015). By defining reduction rules, a grammar, and typing rules, the behaviour and syntax of a language can be fully modeled in Redex. This allows for the evaluation of modeled languages by using Redex's inbuilt functions to type check programs and confirm how said programs would behave using the implemented model of the language. Through these simulations, the desired properties, or lack thereof, of a language can be confirmed. Klein and Findler (2009) is one example of research into this type of testing for intended and unintended behaviour for languages in Redex.

Not only is Redex useful for giving an informal idea of language behaviour, it is also useful for formally designing and proving programming languages. There are many ways in which Redex enables this design and proof process. First, by actually allowing example programs to be run and verified using a model of the language, the process of generating intuition for the features of a language is vastly simplified. This allows for the testing of language features and edge cases to be accomplished easily by leaving complicated proof trees to be handled by Redex. Furthermore, Redex provides a convenient way through which language extensions can be immediately implemented. This streamlines the efforts taken when attempting to add features to an existing language by hand. Thus, these advantages informed our decision to use Redex modeling in our proposed approach to safety proofs in programming languages research.

In surveying relevant research it should, thus, not be surprising that Redex has already been proposed as a tool which can aid in the research cycle for programming languages research. However, most of these papers which use Redex solely focus on the modeling of languages without considering some of the benefits which modeling could bring to proving language safety or other aspects of programming languages research. For example, Klein et al. (2012) incorporates Redex as a tool in prototyping and writing up designs of languages. Additionally, other relevant papers such as Blanco et al. (2019) and Klein (2009) tend to use Redex as a testing tool. But, there does not seem to be any paper which actually attempts to use a Redex based approach to improve the process of proving safety for programming languages. We hope to fill this research gap through our approach.

2.2 Effect Systems

Due to System C being a language built around an effect system, we give a brief overview of effect systems and effects in general. Effects are essentially parts of a program which has side-effects other than just accessing and returning the value of an argument (Turbak and Gifford, 2008). Put differently, effects can be thought of as any piece of code which can not be fully expressed using lambda calculus (i.e., not fully functional). Building on this idea, effect systems are a way through which effects can be reasoned about and described in a computer program (Lucassen and Gifford, 1988). These systems, while somewhat abstract are very useful in controlling and reasoning about the behaviour of a program. This is especially necessary because effectful programs are often harder to understand and keep track of than programs which only read and return values without side-effects.

Currently, there are a suite of languages that implement support for effects whether fully or partially. A prominent example of a language which has support for effects is Haskell (Orchard and Petricek, 2014). Moreover, there are also a myriad of research languages such as Effekt (Brachthäuser et al., 2020), Eff (Kiselyov and Sivaramakrishnan, 2018) and Wyvern (Nistor et al., 2013; Mackay et al., 2019; Potanin et al., 2004; Kurilova et al., 2014) which are built around the core idea of supporting effect systems. In terms of more mainstream languages, Java implements an effect system through its checked exceptions (Gosling et al., 2005). This checked exception mechanism is rather simplistic because it only deals with a generic `throws` effect that can be handled via a handler which is either predefined or explicitly created by the programmer. Interestingly, this type of effect system where effects are able to be handled is similar to the idea of algebraic effects which are a cornerstone of System C.

2.2.1 Algebraic Effects

Ever since the introduction of effect systems in Lucassen and Gifford (1988), there have been a multitude of ways in which programming languages have attempted to implement effect systems. One method of expressing effects which has been gaining popularity recently is that of algebraic effects (Plotkin and Pretnar, 2009). JavaScript, which is often used to create web applications that have asynchronous operations, even has a proposal for the inclusion and support of algebraic effect systems for JavaScript (Macabeus et al., 2020). Algebraic effects, like other effect systems, allows for effects to be tracked throughout a piece of code or program. However, the interesting behaviour of algebraic effect systems is that the systems allow effects to be thrown and handled individually via a handler. This is as opposed to just reasoning about effects without being able to actively handle them.

As mentioned previously, this behaviour of being able to throw and handle effects is similar to the way in which Java deals with exceptions. However, a critical difference is that many implementations of algebraic effect systems allow for these effects to be recovered directly from where the effect was thrown (Lindley et al., 2024). This differs from Java and means that algebraic effect handlers can be more versatile. In particular, asynchronous applications often benefit from the ability to recover from effect being handled (Barga et al., 2004).

2.3 Capabilities

Interestingly, System C tracks effects as capabilities instead of having them as a part of the type of a program. Capabilities are authorities given to objects which allow the object to affect a specific resource (Dennis and Van Horn, 1966). This mechanism allows for more fine-grained control over which objects can interact with resources. A popular example to demonstrate the usefulness capabilities is a logger object. In a language which respects capabilities, a logger would have to be instantiated with a capability to read and append to files. Without this capability set, the logger object would not have access to the ability to read or write to a file object. The potential of capabilities to control resource access is applicable to many situations ranging from security to type checking.

There have been multiple attempts to use capabilities as a way to reason about effects. The Wyvern language successfully bound effects to capabilities to implement type-checking which also reasons about effects (Craig et al., 2018). System C, which we will be extending in our case studies, also implements effects by storing effects as a capability set C. In fact, the interplay between capabilities and effects in System C results in many benefits such as ergonomics and also type safety (Brachthäuser et al., 2022).

2.4 System C

Now that we have established some of the core mechanisms that are needed to understand System C, we will start to describe the actual System C language itself. System C is a library of the Scala language that is extended with an algebraic effect system. This algebraic effect system has handlers which can define the behaviour for dealing with effects at a site in which they occur. Specifically, the `try (code ...) with (handler ...)` control structure is used to run code with a defined handler to manage any effects which may exist as a result of the code which is run.

2.4.1 Comparison Between System C and a Popular Effect Handler

System C, as an algebraic effect system, enables the reasoning of effects in programs while also allowing for more control over how effects are handled. To explain how this algebraic effect system functions, we give a comparison to Java's checked exception handling system. We will use the terms 'effect' and 'exception' interchangeably when we refer to Java because Java's checked exceptions are analogous to effects in other languages.

In Java, checked exceptions are handled by `try (code ...) catch (handler ...)` control structures. Immediately, we can already see parallels between the syntax of System C's handler and Java's handler. Further, the behaviour of the two handlers are also comparable. When an exception is thrown in Java, the handler defined in the `catch` block will process the exception by running code which has been specified by the developer. The same mechanism is present in System C's algebraic handler structure. However, there are also a few unique differences in how handlers behave in System C as opposed to Java.

One large difference between Java and System C is that Java requires explicit type declarations of effects which may be thrown when they are dealt with by a higher-level handler. These are clearly present in the type signature of methods such as in the example of Figure 2.1 where the exception `Exception` is clearly labelled. This behaviour differs from System C in that System C requires no explicit type declarations due to effects being expressed as capabilities.

```
public void negative(int i) throw Exception {
    if (i > 0) {
        throw new Exception();
    }
}
```

Figure 2.1: Explicit Type Declaration in Java

The flexibility to not explicitly declare the types of effects provides ergonomic and practical advantages when writing programs in System C. For one, it is bothersome for developers to always have to list out the effects which a particular program may trigger. Thus, being able to just leave it to the effect system to track is a quality of life improvement. More importantly, there may be occasions in which declaring effects in the type signature is not feasible. One example of this, in Java, is code generated by external tools that are not able to track effects. Having to explicitly declare effects in the type signature would render these tools and libraries unusable.

Another large difference between Java's handler and System C's handler has to do with the actual behaviour of the handler as control flow structure. The conventional wisdom for handlers in popular languages is that code execution will resume after the block of code which caused the effect. However, System C's design allows the ability for code execution to continue from the point at which the effect was triggered. This results in benefits for applications which can not afford for control to be unrecoverable after handling an effect. Some examples where this functionality may be pertinent include web applications and also real-time systems where it may be necessary to continue on in a process even if a failure or effect is detected.

Finally, the last major difference concerns the scoping of the effect handling. Scoping refers to the region within which an identifier (e.g., variable, function, etc.) is accessible (Gazi, 2024). Scoping is usually categorised as either dynamic or static. Dynamic scoping refers to scoping where a "binding is in effect during the whole reduction of an expression to a value" (Tanter, 2009). Contrarily, static or lexical scoping refers to when the scope of a binding is just a programmatic block. Static scoping is how programming languages are usually scoped. Fig 2.2 shows an example of dynamic scoping where the application returns an output of `False` instead of the usual `True` that would be returned if the program were statically scoped.

```
variable = True

return_global():
   return variable

# The dynamic scoping means that ‘variable’ becomes False in the evaluation
# context instead of True
dynamic_override():
   variable = False
   return return_global()

# main() prints out False instead of True because of dynamic scoping
__main__():
  print(dynamic_override())
```

Figure 2.2: Dynamic Scoping Output

The difference in scoping lays in the fact that System C's effects are strictly statically scoped. That is to say, the handler for any given effect in System C is able to be reasoned about in an intuitive manner where each effect is strictly bound to the specific handler which it is defined to be in the scope of. This is in contrast to Java's exception handler which employs both dynamic and static scoping which can cause the exception handling to become less intuitive to reason about (The University of Texas at Austin, 2008). Thus, System C's static scoping helps to mitigate confusion when reasoning about effects and effect handling.

2.4.2 Language Features

Having given some intuition as to System C's main effect handler mechanism through a comparison to Java, we now more intimately describe some of the other features of System C:

Effects as Capabilities. As discussed earlier in the comparison to Java, System C does not require explicit typing of effects in the type signature of a block of code in which the effects appear. This is because of the ability for effects to be represented as capabilities sets which are automatically tracked throughout the evaluation of a program. Thus, effect safety is guaranteed by restricting access to capabilities instead of to the specific effects which each capability may have access to. This is done primarily through making functions second-class.

Second-class Functions. Functions and values within a language can be defined as either first or second class. The distinction between a first class and second class function is determined by the ability for said function to be passed in as a parameter to another function (Abelson and Sussman, 1996). While first class functions can act as arguments to other functions, second class functions can not. Therefore, in the context of effects as capabilities, this effectively neuters the ability of any second class function/capability to allow effects to escape passing the effects as arguments to something which is out of scope. Indeed, in System C, all functions apart from those which are boxed are second class. This boxing mechanism in conjunction with the second class nature of functions is used to ensure effect safety for applications. We expand upon this in Chapter 7.

Boxing. While only having second class functions makes effect safety much easier to guarantee, it is not convenient for writing applications because said functions can not be easily passed around. Therefore, boxing is required to enable the functionality of first class functions. Boxing is a syntactical construct which enables second class functions to become first class by making the implicitly tracked capabilities explicitly defined in a function's type. Figure 2.3 demonstrates an example of converting a second class function f into a first class function. As demonstrated, a **box** construct is used to express boxing and any tracked capabilities are expressed in the type signature explicitly as C. Thus, we go from a second class function f with implicit capabilities C to a first class function f which is explicitly typed as σ **at** C where C contains the previously implicit capabilities/effects.

Although boxing allows second class functions to be passed around, an important prop-

$$f :^{C} \sigma \vdash \textbf{box } f : \sigma \textbf{ at } C$$

Figure 2.3: Boxing a second-class function f

erty of boxed functions is that we may only use them after unboxing. This is important because it prevents capabilities and effects from escaping and being used even when they are out of scope. Figure 2.4 demonstrates the unboxing construct. It is important to note that this unboxing is only valid when all tracked capabilities are still in scope. In Figure 2.4, this is demonstrated by the $f :^{C} \sigma$ being present in the program environment. This is how effect and type safety is maintained even when first class function behaviour is present in System C.

$$\Gamma, f :^{C} \sigma \vdash \textbf{unbox } (\textbf{box } f) : \sigma \mid C$$

Figure 2.4: Unboxing a function

2.4.3 Language Syntax and Rules

To conclude the overview of System C, we present a formalised version of its syntax and rules. Additionally, we explore some aspects of System C's typing and evaluation rules. We start with the language syntax which is displayed in Figure 2.5. As shown in the figure, the language itself is quite simplistic due to the fact that it is mainly designed to demonstrate an implementation of a novel effect system rather than being a fully flesh out, commercial language.

However, that is not to say that nothing can be accomplished using System C. Using this grammar, we can write programs that take the form of s statements. For instance, some valid programs could take the form of:

$$\textbf{val } x = \textbf{return } 0; \textbf{return } x$$

or just simply

$$\textbf{return } \text{true}$$

The syntax provided in Figure 2.5 follows the standard conventions for defining language syntaxes. Each symbol corresponds to a separate syntax element and can be replaced by any corresponding expression on the right side of the assignment. For example, an expression e can take the form of x, box b or any of the primitives at will. Thus, by using the syntax provided in Figure 2.5, any program which is built is semantically valid under the System C language.

On the other hand, the syntactic rules in Figure 2.6 function as rules which allow a

Syntax:				
Expressions	e	$::=$	x	expression variables
		$\mid$	$() \mid 0 \mid 1 \mid \ldots \mid \text{true} \mid \text{false} \mid \ldots$	primitives
		$\mid$	$\textbf{box } b$	box introduction
Blocks	b	$::=$	f	block variables
		$\mid$	$\left\{\left(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j} \Rightarrow s\right)\right\}$	block implementation
		$\mid$	$\textbf{unbox } e$	box elimination
Statements	s	$::=$	$\textbf{def } f = b;\ s$	block definition
		$\mid$	$b\left(\overrightarrow{e_i}, \overrightarrow{b_j}\right)$	block application
		$\mid$	$\textbf{val } x = s;\ s$	sequencing
		$\mid$	$\textbf{return } e$	returning
		$\mid$	$\textbf{try } \{f \Rightarrow s\} \textbf{ with } \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s\right\}$	handlers
Types:				
Value Types	τ	$::=$	$\text{Int} \mid \text{Boolean} \mid \ldots$	base types
		$\mid$	$\sigma \textbf{ at } C$	boxed block types
Block Types	σ	$::=$	$\left(\overrightarrow{\tau_i}, \overrightarrow{f_j : \sigma_j}\right) \to \tau$	
Capabilities	C	$::=$	$\varnothing \mid \{f\} \mid C \cup C$	
Environments:				
Environments	Γ	$::=$	$\varnothing$	empty environment
		$\mid$	$\Gamma, x : \tau$	value bindings
		$\mid$	$\Gamma, f :^{*} \sigma$	tracked bindings
		$\mid$	$\Gamma, f :^{C} \sigma$	transparent bindings

Figure 2.5: Syntax of System C

program on the left of the arrow to step or evaluate into the program on the right of the arrow. These reduction rules are necessary to actually evaluate the programs built using the syntax in a coherent way. As an example, Figure 2.7 demonstrates a reduction in progress from start to finish based off of the reduction rules. Importantly, a reduction rule will always be able to be applied to a valid program until the program becomes a **return** v. Any program which does not fit the constraints of any reduction rule is said to be stuck. The main reason a program would get stuck is due to invalidity of syntax.

But, invalidity of syntax is not the only reason that programs may be invalid. Another reason for programs being invalid stems from incorrect typing of objects. Typing rules, as displayed in Figure 2.9, reason about the types of objects within a language. These typing rules are formatted similar to logical rules and also work in a similar fashion. In order for an object to be well-typed, a valid typing rule tree must be able to be formed. Figure 2.8 is an example of a very primitive typing tree which proves the validity of the program shown. Programs which are not validly typed are often also not able to evaluate properly.

Reduction Rules:

$$
\begin{array}{lll}
(\mathit{box}) & \langle \textbf{unbox}\ (\textbf{box}\ b) \mid \Xi\rangle & \longrightarrow\ \langle b \mid \Xi\rangle \\
(\mathit{val}) & \langle \textbf{val}\ x = \textbf{return}\ v;\ s \mid \Xi\rangle & \longrightarrow\ \langle s[x \mapsto v] \mid \Xi\rangle \\
(\mathit{def}) & \langle \textbf{def}\ f = w;\ s \mid \Xi\rangle & \longrightarrow\ \langle s[f \mapsto w] \mid \Xi\rangle \\
(\mathit{ret}) & \langle \#_l\{\ \textbf{return}\ v\}\ \textbf{with}\ h \mid \Xi\rangle & \longrightarrow\ \langle v \mid \Xi\rangle \\
(\mathit{app}) & \langle ((\{(\overrightarrow{x_i}, \overrightarrow{f_j}) \Rightarrow s\})(\overrightarrow{v_i}, \overrightarrow{w_j}) \mid \Xi\rangle & \longrightarrow\ \langle s[\overrightarrow{x_i \mapsto v_i}, \overrightarrow{f_j \mapsto C_j}, \overrightarrow{f_j \mapsto w_j}] \mid \Xi\rangle \\
 & \quad \text{where } \varnothing \vdash w_j : \sigma_j \mid C_j & \\
(\mathit{try}) & \langle \textbf{try}\ \{f \Rightarrow s\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi\rangle & \longrightarrow\ \langle \#_l\{s[f \mapsto \{l\}, f \mapsto \textbf{cap}_l]\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi\rangle \\
 & \quad \text{where } l \notin \mathit{dom}\ \Xi,\ \text{and}\ \vdash f : \overrightarrow{\tau_i} \rightarrow \tau_0,\ \text{then}\ \Xi(l) \coloneqq \overrightarrow{\tau_i} \rightarrow \tau_0 & \\
(\mathit{cap}) & \langle \#_l\{H_l[\textbf{cap}_l(\overrightarrow{v_i})]\}\ \textbf{with}\ h \mid \Xi\rangle & \longrightarrow\ \langle s[\overrightarrow{x_i \mapsto v_i}, k \mapsto \{y \Rightarrow \#_l\{H_l[\textbf{return}\ y]\}\ \textbf{with}\ h\}] \mid \Xi\rangle \\
 & \quad \text{where } h = \{(\overrightarrow{x_i}, k) \Rightarrow s\} &
\end{array}
$$

Figure 2.6: Operational semantics of System C

$$\textbf{val}\ x = \textbf{return}\ 1; \textbf{return}\ x \longrightarrow \textbf{return}\ 1\ (\mathit{val})$$

Figure 2.7: Reduction Example

$$
\textsc{Val}\ \dfrac{\textsc{Ret}\ \dfrac{\textsc{Lit}\ \dfrac{}{\Gamma \vdash 1 : \mathrm{Int}}}{\Gamma \vdash \textbf{return}\ 1 : \mathrm{Int} \mid \varnothing} \qquad \dfrac{\dfrac{x : \mathrm{Int} \in \Gamma}{\Gamma \vdash x : \mathrm{Int}}\ \textsc{Var}}{\Gamma, x : \mathrm{Int} \vdash \textbf{return}\ x : \mathrm{Int} \mid \varnothing}\ \textsc{Ret}}{\Gamma \vdash \textbf{val}\ x = \textbf{return}\ 1; \textbf{return}\ x : \mathrm{Int} \mid \varnothing}
$$

Figure 2.8: Example of Typing Derivation

2.5 Language Safety

Now that we have established the background necessary to understand the approach we propose and also the System C language which we will be conducting case studies on, we will explore the traditional safety process which our approach enhances.

The process of creating any random programming language is actually trivial; all anyone needs to do is define a grammar, reduction rules and typing rules. However, that does not make the defined language safe. Although there has been research into other methods (Patterson et al., 2022), language safety is classically composed of two theorems - progress and preservation (Wright and Felleisen, 1994). The theorem of progress states that for any arbitrary program in a safe language, that program is either a value or it can be evaluated into another program. As for preservation, the theorem states that for any safe language, a program preserves its current type even after being evaluated according to a reduction rule (Pierce, 2002). These two theorem of progress and preservation make sure that safe languages function exactly as expected from people who use them to write applications.

There are generally two ways in which a researcher can go about attempting a proof of language safety. Either the formal proof of safety can be done unassisted or a mechanised theorem prover can be used to assist in the process of constructing the proof. In fact, theorem provers are tools which enable researchers to proves theorems via computer programs (Nawaz et al., 2019). However, theorem provers also have their distinct advantages and disadvantages when compared to doing proofs unassisted. Chiefly, theorem provers such as Coq (Team, 2024) often suffer from the complexities of the mechanised proving process. As such, simple or even trivial proofs which are easily understood on paper can become difficult to prove when using a theorem prover. Where Coq and other relevant provers do shine is in the fact that any proof extracted from the prover will be guaranteed to be rigorous and mathematically correct.

On the other hand, unassisted proofs of safety allow researchers to leverage preexisting axioms without necessary having to prove them formally. Furthermore, any well-known properties can often be hand-waved without getting bogged down in the details when said properties are intuitively or trivially obviously. That is not to say that non-assisted proofs of safety do not have their own downsides. For one, this way of doing safety proofs is prone to human error which is harder to detect than when a prover is used. Moreover, any changes which are made to the language retroactively may cause problems that propagate throughout the proof of safety. These errors could be caught easily if a prover were being used, but a human researcher may have trouble catching issues that retroactive changes create.

In summary, mechanical provers are often too tedious while non-assisted proofs are prone to errors. This is why the authors of this paper decided to create an approach for proving safety which does not suffer from the pitfalls of only using a mechanical prover or solely doing the proof unassisted. As highlighted, we leverage Redex to model a language before attempting to prove it. Although this modeling is not indicative of language safety, it does help in gaining intuition and also error detecting before an actual safety proof is attempted. We will discuss this further in the next chapter.

Block Typing. $\boxed{\Gamma \vdash b : \sigma \mid C}$

$$\frac{f :^{C} \sigma \in \Gamma}{\Gamma \vdash f : \sigma \mid C}\;[\textsc{Transparent}] \qquad \frac{f :^{*} \sigma \in \Gamma}{\Gamma \vdash f : \sigma \mid \{f\}}\;[\textsc{Tracked}]$$

$$\frac{\Gamma,\ \overrightarrow{x_i : \tau_i},\ \overrightarrow{g_j :^{*} \sigma_j} \vdash s : \tau \mid C \cup \overrightarrow{g_j}}{\Gamma \vdash \{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j : \sigma_j}) \Rightarrow s\} : (\overrightarrow{\tau_i}, \overrightarrow{g_j : \sigma_j}) \to \tau \mid C}\;[\textsc{Block}]$$

$$\frac{\Gamma \vdash b : \sigma \mid C'}{\Gamma \vdash \textbf{unbox}\ e : \sigma \mid C}\;[\textsc{BoxElim}] \qquad \frac{\Gamma \vdash b : \sigma \mid C' \qquad C' \subseteq C}{\Gamma \vdash b : \sigma \mid C}\;[\text{BS}\textsc{ub}]$$

Expression Typing. $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \mathrm{Int}}\;[\textsc{Lit}] \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\;[\textsc{Var}] \qquad \frac{\Gamma \vdash b : \sigma \mid C}{\Gamma \vdash \textbf{box}\ b : \sigma\ \textbf{at}\ C}\;[\textsc{BoxIntro}]$$

Statement Typing. $\boxed{\Gamma \vdash s : \tau \mid C}$

$$\frac{\Gamma \vdash s_0 : \tau_0 \mid C_0 \qquad \Gamma,\ x : \tau_0 \vdash s_1 : \tau_1 \mid C_1}{\Gamma \vdash \textbf{val}\ x = s_0;\ s_1 : \tau_1 \mid C_0 \cup C_1}\;[\textsc{Val}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{return}\ e : \tau \mid \varnothing}\;[\textsc{Ret}]$$

$$\frac{\Gamma \vdash b : (\overrightarrow{\tau_i}, \overrightarrow{f_j : \sigma_j}) \to \tau \mid C \qquad \overrightarrow{\Gamma \vdash e_i : \tau_i} \qquad \overrightarrow{\Gamma \vdash b_j : \sigma_j \mid C_j}}{\Gamma \vdash b(\overrightarrow{e_i}, \overrightarrow{b_j}) : \tau[\overrightarrow{f_j \mapsto C_j}] \mid C \cup \overrightarrow{C_j}}\;[\textsc{App}]$$

$$\frac{\Gamma \vdash b : \sigma \mid C' \qquad \Gamma, f :^{C'} \sigma \vdash s : \tau \mid C}{\Gamma \vdash \textbf{def}\ f = b;\ s : \tau \mid C}\;[\textsc{Def}] \qquad \frac{\Gamma \vdash s : \tau \mid C' \qquad C' \subseteq C}{\Gamma \vdash s : \tau \mid C}\;[\text{SS}\textsc{ub}]$$

$$\frac{\Gamma,\ f :^{*} \overrightarrow{\tau_i} \to \tau_0 \vdash s_1 : \tau \mid C \cup \{f\} \qquad \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \vdash s_2 : \tau \mid C}{\Gamma \vdash \textbf{try}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} : \tau \mid C}\;[\textsc{Try}]$$

Figure 2.9: System C Typing Rule

Chapter 3

Methodology

Because the main motivation behind this thesis is to analyse the approach of leveraging Redex modeling as an intermediary step between designing and proving new language features, case studies are employed to evaluate the performance of the approach. However, we also wish for the case studies which are done to have some tangible value. Thus, we now present the methodology under which we will perform the analysis of the approach while also ensuring that our case studies are sufficiently interesting.

3.1 Redex Based Approach

Let us formally introducing the novel approach which we will explore and analyse within this thesis. Although Redex lies at the heart of the novel approach through which we attempt to conduct our programming languages research, our approach starts similarly to traditional programming languages research. We first define a syntax, typing rules and reduction rules for a language which we wish to explore. Subsequently, Redex is used to implement the language or language feature which we are doing research on. In this way, we can leverage Redex's ability to run and type check example programs to help confirm intuition for the behaviour of our language. Additionally, the capability to run examples also provides a practical test of correctness. Thus, we can preemptively catch some errors in the language feature which is being studied or implemented before a full proof is attempted. Furthermore, this process would allow extensions to the language to be tested and refined in a more fluid manner then provers or standard proofs. This is because having a Redex model of a language allows for continuous iteration within the research process while still maintaining a certain standard of correctness. Finally, the final step of our approach is a proof of safety. This proof of safety should be greatly enhanced by our model of the language which we seek to prove.

To summarise, our approach consists of the following steps:

1. A new language or language feature is designed.
2. A model of the language or language feature is created in Redex. This helps to catch errors in safety before a safety proof is conducted.
3. A proof of safety is conducted.

3.2 Exploring Our Approach

3.2.1 Base System C Redex Model

To begin our exploration into applying a Redex based approach to programming languages research, we first model the original System C language within Redex. Modeling or prototyping is a great way to better understand a language in both its execution and syntax. Furthermore, outside of this paper, System C has not been modeled in Redex. This prototype of System C mainly helps in the case studies which we use to evaluate our approach. By providing a foundation upon which the case studies can build on, we ensure that more robust and realistic case studies are performed. This is because we posit that when researchers attempt to research and prove properties for a language or language feature, they will attempt this proof on the entire language. If this is not the case, mechanisms presented in the novel language feature which is being researched may unknowingly interact with the foundational programming language in a way which is unsafe. We discuss this Redex prototyping in Chapter 4.

Furthermore, we use this part of the thesis to investigate some of the techniques used in modeling language features in Redex. Additionally, Chapter 4 contains an exploration of some of the behaviours of System C under the lens of a Redex implementation. As such, we allow readers of this thesis to gain more intuition behind the mechanisms present in System C. We reinforce this understanding by showing example programs which type check validly to demonstrate intended language behaviour in addition to programs which do not type check. This dichotomy should highlight the way in which System C is designed to function.

3.2.2 Case Studies

Having implemented a base model upon which we can build upon, we now run case studies to investigate the efficacy and practicality of our suggested approach. The two case studies which are attempted both have tangible motivations which are more sophisticated than choosing extensions whimsically. During the two case studies in Chapter 5 and Chapter 6, we also discuss the process through which we extend the base System C Redex model. Moreover, some examples of valid and invalid programs are investigated.

Variables

Our first case study involves variables, a basic language feature which is implemented in most popular languages. This case study is motivated by previous efforts to combine the

System C language with Wyvern. Ou (2023) attempted to combine the System C and Wyvern languages by started with Wyvern before extending the language with elements from System C. Although Ou (2023) managed to produce a system which merged the two languages, the authors did not manage to completely prove the safety of the new, combined language. In particular, variables in the merged language were a hurdle in the safety proof because they produced unsafe behaviour when interacting with System C's effect system. With Ou (2023) serving as motivation, this thesis attempts to merge extend System C with variables through a different approach. We start with the System C language and extending it with variables instead of attempting to extend Wyvern with System C's effects system and variables. Thus, any further work which aims to combine Wyvern with System C can directly build upon our extension to System C rather than extending Wyvern.

Handler Interception

The second case study which is presented in this thesis is an attempt at abstracting the algebraic effects handler mechanism within System C. This extension is primarily motivated by Ouyang (2023), a previous work written by the author. In the aforementioned paper, an attempt was made at combining abstract and algebraic effect systems. However, this did not produce much interesting or emergent behaviours. As such, we take this chance to use this second case study to explore a potentially more meaningful way of abstracting algebraic effects. Once again, we perform this case study under the lens of our novel approach.

3.2.3 Safety

Under our approach, the Redex modeling in the two case studies conducted would have identified any outstanding errors in the safety of our extensions to the System C language. Therefore, we now continue with the second part of our approach by formally proving safety for our whole language. Due to the nature of our Redex modeling, we can also leverage the intuition we gained from implementing our language in Redex while formally proving our language safety.

The proofs of progress and preservation which make up a proof of safety are presented in Chapter 7. This proof will involve the whole System C language including the extensions which we made in the two case studies. Furthermore, this formal safety proof will signal the end of our approach to effect systems research. By using our approach for research, we end up with a formal proof of safety, new language features or effect system mechanism, and a usable model of the language which performed research on.

3.3 Discussion

To conclude our investigation into our proposed Redex augmented approach, Chapter 8 is a discussion about the entirety of our novel approach. We discuss the efficacy and practicalities of using Redex in programming languages research. Furthermore, we informally compare our case studies against a regular research cycle and highlight differences which were produced by using Redex for modeling. This chapter also includes a brief overview of limitations for our approach and any overall sentiments. Finally, to round off, we present some ideas for applications and benefits of the extensions which were implemented in the case studies.

Chapter 4

Base System C Model

Before we can conduct any case studies under our Redex based approach, we need to develop a foundational model of System C which can be extended. By modeling the language, we will also be able to more intimately understand the grammar, reduction rules, environments and any other interesting components of System C. Furthermore, we will also be able to expand on how modeling is accomplished using Redex. This understanding will provide context for any further modifications which are made to System C in the case studies. Finally, any decisions which differ from the original Brachthäuser et al. (2022) paper are also explored.

4.1 Grammar

To create an accurate model of System C, we first need to start with modeling the grammar of the language into Redex. More specifically, our objective is to make sure that any programs which are possible in System C should be allowed under our Redex implementation and any programs which are not syntactically accurate should be disallowed. In practical terms, this will involve modeling all the components presented in Figure 2.5.

To implement our grammar within Redex, we use the in-built `define-language` function. This function works to define a context-free grammar where the first symbol can be replaced by any other symbol which is defined after it (Cremers and Ginsburg, 1975). This is essentially how the syntax in Figure 2.5 is expressed anyhow. Thus, if we take the example of:

$$e$$

According to the defined grammar in Figure 2.5, e can be replaced by `x`, `()`, `natural`, `true`, `false` or `box b`. In the first block of our modeling of the entire System C syntax in Figure 4.1, we can indeed see that an expression e can be replaced by any of the other valid terms listed above. Figure 4.1 shows the entire one-to-one mapping of

```
(define-language System_C
  (e x
     ()
     natural
     true
     false
     (box b))

  (b f
     ((x : τ) ... , (f : σ) ... ⇒ s)
     (unbox e))

  (s (def f = b ; s)
     (b (e ... , b ...))
     (val x = s ; s)
     (return e)
     (try f ⇒ s with h))

  (τ Int
     Boolean
     (σ at C))

  (σ (τ ... , (f : σ) ... → τ))

  (C (f ...))

  (Γ (g ...))

  (g (x : τ)
     (f :* σ)
     (f : C σ))

  (x variable-not-otherwise-mentioned)

  (f variable-not-otherwise-mentioned))
```

Figure 4.1: Redex Implementation of Basic System C Grammar

System C's grammar (Figure 2.5) to a Redex implementation.

Whilst most of the code in Figure 4.1 is a one-to-one translation from the written grammar to the modeled one, there are a few interesting points worth mentioning. In particular, ellipses (`...`) in Redex represent zero or more repetitions of a particular term. This is synonymous with the effect of curly brackets ({}) within a Backus–Naur form (BNF) grammar (McCracken and Reilly, 2003). As an example, the production rule in

Figure 2.5:

$$\left\{\left(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j} \Rightarrow s\right)\right\}$$

is represented by:

```
((x : τ) ... , (f : σ) ... ⇒ s)
```

where a $\overrightarrow{x_i : \tau_i}$ is symbolised by a `(x : τ) ...` term.

Furthermore, to prototype both the capability sets C and the environment Γ of a language, we take advantage of Redex's built-in list functionality. Incidentally, ellipses are also how lists are defined in Redex. In order to define lists which only contain elements of the type that we want, we need to first create terms which represent said wanted elements. For instance, a list of x's would be defined as `(x ...)`.

This unique way of defining lists is most important when we consider the environment Γ in System C. This is because, as according to the syntax in Figure 2.5, Γ is a list which contains value bindings $x : \tau$, tracked bindings $f :^{*} \sigma$, and transparent bindings $f :^{C} \sigma$. Thus, to work around the way Redex defines lists, we represent Γ by creating an intermediate term `g` which reflects the three types of terms which an environment Γ could contain. Therefore, we define Γ as:

```
(g ...)
```

where `g` is:

```
(g (x : τ)
   (f :* σ)
   (f : C σ))
```

This allows Γ to be defined properly via Redex's rules while also keeping the ability for multiple bindings to be present in Γ as opposed to just one type of binding.

We now move on to the capability set. Figure 2.5 specifies that a capability set is:

$$C ::= \varnothing \mid \{f\} \mid C \cup C$$

In Redex, we represent a capability set as a list which can only contain `f`s. To address the issue of potential duplicate elements within this list, we make sure to only add `f`s to our Redex capability set if we are sure that the `f` does not already exist within our capability set.

Finally, the last elements of interest in our Redex modeling of the System C grammar are the terms `x` and `f`. In Figure 4.1 it can be seen that we defined `x` and `f` by the Redex term `variable-not-otherwise-mentioned`. This is because `x` and `f` are expression and block variables respectively. As such, they can be defined as any name as long as it is not occupied by another existing term. Luckily, this is exactly what the `variable-not-otherwise-mentioned` term expresses in Redex.

4.2 Typing Rules

Now that we have established a Redex implementation of the syntax for System C, we will begin to model the rules for type checking any programs which may be formed from the grammar. Similar to the syntax, we refer to the typing rules provided in the Brachthäuser et al. (2022) paper and Figure 2.9 for the modeling.

We will approach the rules by splitting them into three distinct categories of typing rules. Each category will represent a different type of grammatical structure which can be 'typed' by a rule. These are represented in the boxes in Figure 2.9 as block types, expression types, and statement types. The grammatical structure which correspond to the three categories are $b : \sigma$, $e : \tau$, and $s : \tau$ respectively.

However, before we start with explaining the typing implementing, we describe some of the common elements about how typing rules are defined in Redex. To start, the in-built Redex function `define-judgment-form` is used to define a typing rule. This limits each typing rule to only apply when the syntax of of program matches the form wanted by the typing rule. The exact form which a rule expects is defined within the `#:contract` and `#:mode` definitions above each typing rule.

Each square bracket (`[]`) block within a `define-judgment-form` function defines a typing rule. As an example, the Transparent rule is the first typing rule defined within Figure 4.3 and it is encased within square brackets (`[]`). The statements above the dotted line are preconditions for a rule while the statements below the line are post-conditions which are guaranteed by the rule. Continuing with the Transparent rule example, the preconditions would be that $f \in \Gamma$ and C is a subset of c. Because most of the Redex implementation is trivial, we will only explore any interesting aspects of the typing rule modeling.

Before we start exploring any specific typing rule, Figure 4.2 highlights any helper functions which are used in our modeling of the typing rules.

`where`	A function which holds if the output of a function is the type specified.
`find`	Returns an element from a list.
`subset`	Returns true if an a set is a subset of another set.
`set-append`	Appends two sets together and returns the resulting set.
`set-minus`	A function which performs the set minus operation.
`expr-type`	The expression typing rules.

Figure 4.2: Block Type Functions

Block typing rules. We start with the block typing rules. The Redex code for these typing rules are expressed in Figure 4.3. The contract of the block typing rules

```
(block-type Γ b σ c C)
```

mirrors that of the actual form of the block typing rules

$$\Gamma \mid \Sigma \vdash b : \sigma \mid C$$

However, there are a few major differences between the Redex model and the original block typing rules. First, for convenience, any syntactic sugar such as '|' and '⊢' has been omitted from the Redex implementation. This should not affect the functionality of the model because the syntactic sugar is mainly for human readability. The removal of syntactic sugar is assumed throughout all of the typing rules.

The other, more significant, change is that the contract contains an extra capability set represented by the lowercase `c`. This capability set is defined as:

```
(c None
   C)
```

To explain why there exists an extra capability set within the form of the typing rules, we must first examine the `#:mode` tag of Redex judgment forms. In conjunction to the grammatical form of the typing rules specified by the `#:contract` tag, we must also define whether each element in the rule is of an input type or an output type. When an element of a typing rule is an input, that particular term is expected to be known and given to the typing rule. On the other hand, when a term is defined to be an output, that term will be determined after a typing rule is applied.

To illustrate this point, imagine that we are given a term $1 : \tau$. In this instance, we know the input 1. Thus, it is an input term. However, we need to apply a typing rule to $1 : \tau$ to be able to determine the exact form of τ. Therefore, τ would be an output. Under the lens of the block typing rules, `Γ`, `b` and `c` are input types while `σ` and `C` are output types.

When typing rules are usually designed this input and output relationship between terms is often implicit. As such, terms may act as both an input and output at the same time. This is exactly the situation for the capability set C within the typing rules for System C. Specifically, due to the substitution rule BSub in Figure 4.4, we need to know what exactly our capability set C' could be because it may be a subset of another capability set C. As such, we constrain to C be an input term. However, we also know that a capability set C is generated and updated every time we apply a typing rule. Thus, by necessity, C must also be an output term. That is why we have the two capability sets `c` and `C` in our Redex model of the typing rules.

Expression typing rules. The expression typing rules are displayed in Figure 4.5 and do not contain any non-trivial components. Thus, we will omit repeating the general process through which the typing rules are created.

Statement typing rules. As shown in Figure 4.8, Redex's implementation of the statement typing rules are the largest category of typing rules. However, similar to the expression typing rules, there is not much non-trivial content to discuss. Most of the

```
(define-judgment-form System_C
  #:contract (block-type Γ b σ c C)
  #:mode (block-type I I O I O)

  [(where (C σ) (find f Γ))
   (where #t (subset C c))
   ------------------------ "Transparent"
   (block-type Γ f σ c C)]

  [(where (* σ) (find f Γ))
   (where #t (subset (f) c))
   ------------------------- "Tracked"
   (block-type Γ f σ c (f))]

  [(statement-type (g ... (x : τ₁) ... (f₁ :* σ) ...) s τ \
   (set-append c (f₁ ...)) C)
   (where #t (subset C (set-append c (f₁ ...))))
   ---------------------------------------------------- "Block"
   (block-type (g ...) ((x : τ₁) ... , (f₁ : σ) ... ⇒ s) \
   (τ₁ ... , (f₁ : σ) ... → τ) c (set-minus (f₁ ...) C))]

  [(expr-type Γ e (σ at C))
   (where #t (subset C c))
   ------------------------------ "BoxElim"
   (block-type Γ (unbox e) σ c C)]
  )
```

Figure 4.3: Redex Block Typing Rules

$$\frac{\Gamma \mid \Sigma \vdash b : \sigma \mid C' \qquad C' \subseteq C}{\Gamma \mid \Sigma \vdash b : \sigma \mid C}$$

Figure 4.4: BSUB Rule

complexities have been explained within the discussion for the block typing rules and readers can refer to the block typing rules for any complexities within implementation for the statement typing rules.

4.3 Evaluation Context and Reduction Rules

The evaluation context and reduction rules are the final element of the System C language which still needs to be modeled. Evaluation contexts and reduction rules work in tandem to produce deterministic rules which specify the order in which a valid program should

```
  (define-judgment-form System_C
#:mode (expr-type I I O)
#:contract (expr-type Γ e τ)

[
 --------------------------- "Lit"
 (expr-type Γ natural Int)]

[(where τ (find x Γ))
 -------------------- "Var"
 (expr-type Γ x τ)]

[(block-type Γ b σ none C)
 ------------------------------- "BoxIntro"
 (expr-type Γ (box b) (σ at C))]
)
```

Figure 4.5: Expression Typing Rules

be evaluated. Specifically, evaluation contexts identify which part of a program should be evaluated. Take a dummy example:

$$\textbf{val}\ x = \textbf{return}\ 1; \textbf{return}\ x$$

Intuitively, we know that we need to first evaluate **val** x = **return** 1 before **return** x. However, when programs are nested and more complex, using intuition is hard for humans and even more infeasible for machines. As such, the evaluation context encodes the order in which we should start evaluating. For the **val** example, the evaluation context **val** $x = E; s$ shows that the section of the program represented by E should be evaluated before anything else. Overall, Figure 4.6 shows the evaluation context of System C and Figure 4.7 displays the Redex implementation of said context.

Evaluation Contexts:

Contexts	E	::=	$\square$ | **val** x = E; s	| $\#_l$ { E } **with** { $(\overrightarrow{x_i},\ k) \Rightarrow s$ }	
Delimited Contexts	H_l	::=	$\square$ | **val** x = H_l; s	| $\#_{l'}$ { H_l } **with** { $(\overrightarrow{x_i},\ k) \Rightarrow s$ }	where $l \neq l'$

Figure 4.6: System C Evaluation Context

The Redex implementation of evaluation contexts is quite trivial; each evaluation context just has to be declared when we are defining the System C grammar. It may be noted that our implementation of the evaluation contexts does not include the delimited contexts. This is for good reason because the delimited contexts are actually not necessary for full functionality of the System C language. Instead, the delimited contexts are auxiliary contexts used primarily for the proof of safety for System C. As such, we decided to omit

```
(E hole
  (val x = E ; s)
  (l E with (x ... , k) ⇒ s)
```

Figure 4.7: Redex System C Evaluation Context

said delimited contexts because they do not add anything to our implementation.

To actually use evaluation contexts, they need to be considered in conjunction with the reduction rules as presented in Figure 6.9. Figure 4.9 shows the Redex implementation of the reduction rules and how they work in conjunction with the evaluation contexts. The reduction rules present in Figure 4.9 function by evaluating the first program to the second program. Take the following example:

```
(--> (in-hole E (unbox (box b)))
     (in-hole E b)
     "box")
```

This *(box)* reduction rule would imply that any term which matches `E (unbox (box b))` should evaluate to `in-hole E b`. This is where the evaluation context comes into play. As defined in Figure 4.7, `E` represents an evaluation context. Additionally, the `in-hole` function specifies that Redex should try to evaluate anything within an evaluation context as much as possible. Thus, by virtue of already establishing all evaluation contexts `E` and the `in-hole` command, Redex is automatically able to determine a suitable evaluation context. This is one of the advantages of Redex modeling.

Apart from this interaction between the evaluation context and the reduction rules, the remainder of the Redex code is a one-to-one translation from the System C reduction rules in Figure 6.9.

4.4 Examples

To complete our Redex modeling of System C, we have created a list of examples which we used to gain intuition about System C and our model. These examples will include both invalid and valid programs. This is a crucial facet of our novel approach because running Redex examples allows for a more through and robust understanding of the behaviour of language features before any attempt to prove them is made. As such, the list of examples presented can be used as a metric to determine whether our language is correct at a behavioural level. A non-comprehensive list of our examples is provided in Figure 4.10 and Figure 4.11.

```
(define-judgment-form System_C
  #:mode (statement-type I I O I O)
  #:contract (statement-type Γ s τ c C)

  [(statement-type (g ...) $s_0$ $\tau_0$ c $C_0$)
   (statement-type (g ... (x : $\tau_0$)) s_1 $\tau_1$ c $C_1$)
   (where #t (subset $C_0$ c))
   (where #t (subset $C_1$ c))
   ---------------------------------------------------------------- "Val"
   (statement-type (g ...) (val x = $s_0$ ; $s_1$) $\tau_1$ c (set-append $C_0$ $C_1$))]

  [(expr-type Γ e τ)
   ------------------------------------- "Ret"
   (statement-type Γ (return e) τ c ())]

  [(block-type Γ b ($\tau_1$ ... , (f : $\sigma_1$) ... → τ) c C)
   (expr-type Γ $e_1$ $\tau_1$) ...
   (block-type Γ $b_1$ $\sigma_1$ c $C_1$) ...
   (where #t (subset C c))
   (where (#t ...) ((subset $C_1$ c) ... ))
    --------------------------------------------------------------- "App"
   (statement-type Γ (b ($e_1$ ... , $b_1$ ...)) (substitute τ [f $C_1$] ...) c
       (set-append (flatten ($C_1$ ...)) C))]

  [(block-type (g ...) b σ c C_prime)
   (statement-type (g ... (f : C_prime σ)) s τ c C)
   (where #t (subset C_prime c))
   (where #t (subset C c))
   ---------------------------------------------- "Def"
   (statement-type (g ...) (def f = b ; s) τ c C)]

  [(statement-type (g ... (f :* ($\tau_1$ ... , → $\tau_0$))) $s_1$ τ (append f c) C)
   (statement-type (g ... ($x_1$ : $\tau_1$) ... (k : C ($\tau_0$ , → τ))) $s_2$ τ c C)
   (where #t (subset C (append f c)))
   ---------------------------------------------------------------- "Try"
   (statement-type (g ...) (try f ⇒ $s_1$ with (($x_1$ : $\tau_1$) ... , (k : $\tau_0$) ⇒ $s_2$)) τ
       c (set-minus (f) C))]

  [(where ($\tau_1$ ... → $\tau_0$) (find l Γ))
   (where #t (subset (l) c))
   ------------------------------------------------- "Cap"
   (statement-type Γ (cap l) ($\tau_1$ ... , → $\tau_0$) c (l))]

  [(where ($\tau_1$ ... → $\tau_0$) (find l (g ...)))
   (statement-type (g ...) $s_1$ τ (append l c) C)
   (statement-type (g ... ($x_1$ : $\tau_1$) ... (k : C ($\tau_0$ → τ))) $s_2$ τ c C)
   (where #t (subset C (append l c)))
   ------------------------------------------------- "Reset"
   (statement-type (g ...) (l $s_1$ with (($x_1$ : $\tau_1$) ... , (k : $\tau_0$) ⇒ $s_2$)) τ c
       (set-minus (l) C))])
```

Figure 4.8: Statement Typing Rules

```
;; Reduction Rules
(define reduction
  (reduction-relation
   System_C
   #:domain (Γ s)

   (--> (in-hole E (unbox (box b)))
        (in-hole E b)
        "box")

   (--> (in-hole E (val x = (return v) ; s))
        (in-hole E (substitute s [x v]))
        "val")

   (--> (in-hole E (def f = w ; s))
        (in-hole E (substitute s [f w]))
        "def")

   (--> (in-hole E (l (return v) with h))
        (in-hole E (return v))
        "ret")

   (--> (in-hole E (((x_1 : τ_1) ... , (f_1 : σ_1) ... ⇒ s) (v_1 ... , w_1 ...)))
        (in-hole E (substitute s [x_1 v_1] ... [f_1 C] ... [f_1 w_1] ...))
        (judgment-holds (multi-block (w_1 ...) (σ_1 ...) (C ...)))
        "app")

   (--> ((g ...) (in-hole E (try f ⇒ s with ((x_1 : τ_1) ... , (k : τ_0) ⇒
       s_prime))))
        ((g ... (l : τ_1 ... → τ_0)) (in-hole E (l (substitute s [(l) f] [(cap l)
            f]) with ((x_1 : τ_1) ... , (k : τ_0) ⇒ s_prime))))
        (fresh l)
        "try")

   (--> (in-hole E (l (in-hole E_1 ((cap l) (v_1 ... , ))) with ((x_1 : τ_1) ... ,
       (k : τ_0) ⇒ s)))
        (in-hole E (substitute s [x_1 v_1] ... [k ((x : τ_0) , ⇒ (l (in-hole E_1
            (return x)) with ((x_1 : τ_1) ... , (k : τ_0) ⇒ s)))]))
        (fresh x)
        "cap")
   )
  )
```

Figure 4.9: Redex System C Reduction Rules

Program	**Output**
return box ((,) ⇒ return 0)	τ = ((,) → Int) at ()
return 0	τ = Int
def f = unbox box ((,) ⇒ return 0); return box f	σ = ((,) → Int)
def block = ((,) ⇒ return 0); return box block	τ = ((,) → Int) at ()
val boxed = return box unbox box ((,) ⇒ return 0); return boxed	τ = ((,) → Int) at ()
def ret = ((,) ⇒ return 0); return 1	τ = Int
val value = return 1; return value	τ: Int
try f ⇒ (return box ((,) ⇒ val g = return box f; return 42))) with ((, k : Int) ⇒ return box ((,) ⇒ return 0))	τ: ((,) → Int) at ()
def f = ((,) ⇒ return 0); (f (,))	τ: Int Evaluation: (() (((,) ⇒ (return 0)) (,)))
val x = return new 0; return !x	Store: l ↦ 0 Evaluation: val x = return l; return !x
val x = return new 0; val y = x := 1; return !x	Store: l ↦ 0 Evaluation: val x = return l; val y = x := 1; return !x
val x = return new 0; return x := 1	Store: l ↦ 0 Evaluation: val x = return l; return x := 1
val x = return new box ((,) ⇒ return 1); return !x	Store: l ↦ box ((,) ⇒ return 1) Evaluation: val x = return l; return !x

Figure 4.10: Valid Examples in Redex

Program	**Failing Reason**
box unbox 0	Cannot unbox something which is not a block.
return unknown	x is not defined.
return return 0	The syntax for return is incorrect.
try f ⇒ (return box ((,) ⇒ val g = return box f; return 42))) with ((, k : Int) ⇒ return 0)	Continuation is not well-typed.
return new 0	You cannot return a location.
return !0	The variable has yet to be assigned.
val x = return new 0; val y = x := true	Different types while assigning variable

Figure 4.11: Invalid Examples in Redex

Chapter 5

Case Study 1: Adding Variables

To evaluate our novel approach, we perform two case studies which use our Redex based approach. Our first case study involves the extension of variables to System C. Because we undertake this addition to the System C language by utilisng our approach, we first design the syntax, typing rules and reduction rules for the variable construct which we will add before modeling the extended language feature of variables in Redex. The purpose of this case study is to demonstrate how our novel approach would work using a realistic situation and highlight how it may enable researchers to detect errors in safety before a safety proof is attempted.

5.1 Syntax, Typing Rules, and Reduction Rules

5.1.1 Design and Syntax

As specified in the steps laid out in our approach in Chapter 3.1, we start by designing the syntax, typing rules and reduction rules for our new language feature. Because variables are a very common language feature, we draw inspiration from the Types and Programming Languages textbook by Pierce (2002). In particular, the design of variables is derived from that of references from Chapter 13 of Types and Programming Languages. Figure 5.1 presents all new additions to the syntax of System C to enable variables.

The idea of using references to emulate variables works because of how references function. References are literally references to a cell in which information is contained (Pierce, 2002). Thus, we can think of variables as references to a cell which can be changed. Therefore, when we are initialising a new variable, we are creating a new cell in which we can store information. Dereferencing the variable will return the information that is stored within the cell and assigning information to the variable will overwrite the information that is already in the cell.

Syntax:

Expressions	e	$::=$	$\textbf{new}\ e$	initialisation
		$\mid$	$!e$	dereferencing
		$\mid$	$e \coloneqq e$	assignment
		$\mid$	l	store location

Types:

Value Types	τ	$\coloneqq$	$\textbf{Ref}\ \tau$	reference types

Figure 5.1: Variable Extended Syntax of the language

To actually add variables to System C, there are also some factors which need to be considered apart from just the additional syntax. In order to correctly type variables, a new type has to be introduced which is indicative of the fact that variables are references. This is why the **Ref** τ type has been added to the syntax. Additionally, although the user should not directly be able to write to the location of the reference cell, the reference cell location needs to be accessed during the evaluation of a program. Thus, a reference cell location l must be a viable form that an expression e can take.

5.1.2 Typing Rules

We now consider the typing rules which should apply to the new variable constructs which we have introduced. Due to the fact that the addition of variables mainly affect how expressions operate, we only need to consider typing rules which type check expressions. Figure 5.2 presents all the additional typing rules which were needed to type variables within System C and we briefly describe each typing rule below.

$$\boxed{\Gamma \mid \Sigma \vdash e : \tau}$$

$$\frac{\Gamma \mid \Sigma \vdash e : \textbf{Ref}\ \tau}{\Gamma \mid \Sigma \vdash !e : \tau}\ [\textsc{Deref}] \qquad \frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \textbf{new}\ e : \textbf{Ref}\ \tau}\ [\textsc{New}]$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \textbf{Ref}\ \tau \qquad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1 \coloneqq e_2 : \tau}\ [\textsc{Assign}] \qquad \frac{\Sigma(l) = \tau}{\Gamma \mid \Sigma \vdash l : \textbf{Ref}\ \tau}\ [\textsc{Ref}]$$

Figure 5.2: Variable Extended Typing Rules

Deref. Our typing rule for dereferencing specifies that the dereferenced variable will have the type of **Ref** τ. This makes sure that dereferenced variables must be a referenced value before they are dereferenced. Otherwise, there would not be anything to dereference.

New. The New rule states that when creating a new variable through the **new** keyword, the type of the expression τ must be well-typed and that the new variable will have the **Ref** τ type. This variable typing ensures that the Deref rule functions properly.

Assign. The Assign rule ensures that, when assigning a new value to a variable, the value is of the same type as that of the variable. This prevents type mismatching.

Ref. The Ref rule enforces that the type of a reference location l has an associated type of **Ref** τ. This underpins the other typing rules because it ensures that we keep track of variable types in a unique manner which makes them identifiable as variables.

5.1.3 Reduction Rules

Our reduction rules for the variables (Figure 5.3) are also inspired by Pierce (2002). Crucially, every variable's cell location and associated value is stored in Ξ as a key value pair. Similar to the typing rules, we give a brief overview of them below.

(*deref*). Wherever the location of a reference cell is dereferenced, we want to return the value which was stored in the reference cell.

(*assign*). When we assign a new value to a location, we update the location-value pair within Ξ so that when it is dereferenced later, the value will be correct.

(*new*). The creation of a new variable results in the addition of a location-value pair to Ξ. We also make sure that the newly added location does not collide with a preexisting location which is already within the domain of Ξ.

$$
\begin{array}{lll}
(\mathit{deref}) & \langle !l \mid \Xi \rangle \longrightarrow & \langle v \mid \Xi \rangle \\
& \quad \text{where } \Xi(l) = v & \\
(\mathit{assign}) & \langle l \coloneqq v \mid \Xi \rangle \longrightarrow & \langle v \mid \Xi[l \mapsto v] \rangle \\
(\mathit{new}) & \langle \textbf{new } v \mid \Xi \rangle \longrightarrow & \langle l \mid \Xi,\ l \mapsto v \rangle \\
& \quad \text{where } l \notin \mathit{dom}\ \Xi &
\end{array}
$$

Figure 5.3: Extended Reduction Rules

5.2 Redex Modeling

5.2.1 Implementing the Model

Due to the fact that our modeling of the syntax, typing rules and reduction rules is a one-to-one translation from the design previously specified in this chapter, we will not elaborate on much of the modeling process. It should be sufficiently clear from the figures (Figure 5.4, Figure 5.5, and Figure 5.6) which pieces of code correspond to which design elements (i.e., syntax, typing rules, and reduction rules). Instead, we demonstrate some of the safety errors caught because we were able to run examples via our model because this is what our approach contributes to the process of proving safety for a language. The only thing that we will note is that it was simple to add the variables rules as extensions to the base Redex model that was already implemented.

5.2.2 Running Examples and Preemptive Error Detection

We now highlight some of the errors which were discovered by modeling various iterations of our variable extension for System C in Redex. The first issue in the safety of the variable extension had to with expressions. In the original System C language, expressions e are static and can not be evaluated beyond their primitive values. However, with the addition of variables, expressions may now have to be evaluated. This includes situations such as variable assignment, dereferencing and initialisation.

Due to the fact that these operations were not necessary to consider before, the evaluation context of System C does not support operations done by expressions. This can be seen in Figure 4.6 where blocks b are not further evaluated in the evaluation context because there are no operations which would be necessary to evaluate in a block.

At first, our addition of variables to System C did not change any of the evaluation contexts. However, due to some examples that we ran using our Redex model, we discovered that some of the programs we were checking did not output the intended result. One of the example programs which we ran through Redex was:

```
val num = return (new 1); def f = { (x : Int) ⇒ return x }; f(!num)
```

This program is actually quite simple. All it does is assign 1 to a variable `num`. Then, we create a function in System C takes an integer as an argument and returns the integer that is given. Thus, when we ran this example in our Redex model, we expected the model to output 1 because that is what is contained within the `num` variable. However, instead of the expected result, Redex outputted an error because the program got stuck in evaluation. Thanks to this error, we were able to realise that, without an appropriate evaluation context, the assignment of a variable would not be able to be evaluated when we call a function. In essence, the program would evaluate in the following steps:

```
val num = return (new 1); def f = { (x : Int) ⇒ return x }; f(!num)
→ def f = { (x : Int) ⇒ return x }; f(!(new 1))
→ { (x : Int) ⇒ return x }(!(new 1))
→ return (!(new 1))
```

As shown above, Redex tells us that the program would get stuck at `return (!(new 1))` because we no longer can apply any of our reduction rules. As such, we added the new evaluation context

$$b(E..., b...)$$

This evaluation context allows for expressions which are being applied via the *(app)* rule to be further evaluated. After this change, running our example program in Redex now results in an evaluation flow of:

```
val num = return (new 1); def f = { (x : Int) ⇒ return x }; f(!num)
→ def f = { (x : Int) ⇒ return x }; f(!(new 1))
→ { (x : Int) ⇒ return x }(!(new 1))
→ { (x : Int) ⇒ return x }(!l)
→ { (x : Int) ⇒ return x }(1)
→ return 1
```

After we added the new evaluation context, the program was able to evaluate our variable assignment and assign the location of a memory cell to the variable `num`. Thus, instead of being stuck at `return (!(new 1))`, our program was able to evaluate the variable assignment into the value which was contained in the variable cell location. Therefore, our approach of using Redex to model a language feature allowed us to detect this error before the error would have been discovered while doing the proof of safety.

Furthermore, we also used the Redex model to help resolve concerns we had about our language extension. Specifically, a consideration which we had involved the possibility for effects to escape via variables. We were mainly reviewing the possibility of variable nesting (or mutual referencing) allowing effects to escape. For instance, take a variable x with capability f. If another variable y refers to x, it will also have capability f. This example can be continued on for any arbitrary amount of nested variables and, in and of itself, is no problem. The worry is that, even when the original variable x is deleted, the other variables still have access to the capability f. As a result, this would allow f to be accessed even where it may be out of scope.

However, after running a variety of examples in our Redex model, there seemed to be a valid solution to this problem within the original System C design. If a capability is present within an expression, it will have to be boxed. Additionally, a boxed block has a type of σ at C. In theory, this would mean that, even if a variable y is able to refer to capability f outside of the original scope of f, the type of the variable would still be σ at f. As such, when type checking occurs, the program would be invalidly typed due to the fact that f is no longer in scope but is still referenced. This was confirmed by the examples which we ran through Redex and, as such, we were confident that our extension of variables to System C would be valid and safe. Once again, our approach of utilising Redex modeling allowed us to leverage our model of the variable language extension to solve problems we had concerning the safety of our language.

5.3 Safety

The final step of our approach involves a proof of safety. Due to the fact that we attempt two case studies, we defer this proof of safety to Chapter 7 such that we can prove safety for the entire System C language including our two extensions made in the case studies. Moreover, due to the Redex modeling step from our approach, we have already detected and removed many errors that may previously not have been caught until we attempted a proof of safety. Thus, from this case study, we can express that our approach yields benefits before a proof of safety is undertaken.

5.4 Redex Code

```
(e x
   ()
   natural
   true
   false
   (box b)
   (new e)
   (! e)
   (e := e)
   (l))

(τ Int
   Boolean
   (σ at C)
   (Ref τ))
```

Figure 5.4: Variable Model Syntax

```
;; Typing rule for expression typing
(define-judgment-form System_C
  #:mode (expr-type I I I O)
  #:contract (expr-type Γ Σ e τ)

  [(expr-type Γ Σ e (Ref τ))
   ------------------------------- "Deref"
   (expr-type Γ Σ (! e) τ)]

  [(expr-type Γ Σ e τ)
   ------------------------------- "New"
   (expr-type Γ Σ (new e) (Ref τ))]

  [(expr-type Γ Σ e_1 (Ref τ))
   (expr-type Γ Σ e_2 τ)
   ------------------------------- "Assign"
   (expr-type Γ Σ (e_1 := e_2) τ)]

  [(where τ (find l Σ))
   ------------------------------- "Ref"
   (expr-type Γ Σ l (Ref τ))]
  )
```

Figure 5.5: Variable Model Typing Rules

```
   (--> (in-hole E (! l))
        (in-hole E v)
        "deref")

  (--> (in-hole E (l := v))
        (in-hole E v)
        "assign")

  (--> (in-hole E (new v))
        (in-hole E l)
        (fresh l)
        "new")
```

Figure 5.6: Variable Model Reduction Rules

Chapter 6

Case Study 2: Handler Interception

This is our second of two case studies which we implement to demonstrate the efficacy of our approach. In this case study, we use our approach to extend System C by adding a new interception mechanism to the handlers in System C and eventually doing a proof of safety for our extension. Once again, we structure this section according to the steps which are defined in our approach. Specifically, we first design the language extension before modeling the extension in Redex. Finally, we attempt a proof of safety. The aim of this case study is to demonstrate how our novel approach can provide enhance the tradition method of proving of safety in language research by allowing preemptive error detection. At the end of the chapter, we also present the full syntax, typing rules and reduction rules for System C including the extensions we made in the case studies.

6.1 Syntax, Typing Rules, and Reduction Rules

6.1.1 Design and Syntax

Beginning with the first step of our proposed approach, we start to implement the syntax, typing rules, and reduction rules for our extension to the System C language. The main intuition behind this case study's extension to System C is to allow for some abstraction to the algebraic effects handling system which was explained in Chapter 2.4.1. There may be many situations in which this addition to the language could yield benefits. For one, if external code is being run by the handler, the developer may want to take control of the effect handling to ensure that nothing nefarious is being run before either returning control to the original handler or continuing without ceding back control. Additionally, abstraction also helps to hide certain properties of the program either for convenience or security.

Figure 6.1 shows the additions to System C's syntax which allow for the abstraction of the algebraic handler.

$$\text{Statements} \quad s \quad ::= \quad \textbf{intercept}\ \{f \Rightarrow s\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s\right\} \quad \text{handlers interception}$$

Figure 6.1: Abstraction Extension to System C

The structure of the handler interception is inspired by the original System C handler syntax presented in Figure 2.5. This is because the handler interception can be thought of as an additional handler which acts on a preexisting capability f. Because the capability f already exists, the new handler would 'intercept' the effect if it were ever caused. Thus, instead of being handled by the original handler, our new handler would be the one handling the capability f. Due to the fact that our interception mechanism is similar to the original System C handler mechanism, we decided to design our new interception syntactical structures to match the original handling structures. This can be seen in the similarities between our intercept construct and the original handler construct in Figure 6.7.

6.1.2 Typing Rules

Figure 6.2 displays the two additional typing rules which were created to enforce the well-typedness of programs with handler intercepts. We briefly explain the purpose of these two typing rules.

$$\boxed{\Gamma \mid \Sigma \vdash s : \tau \mid C}$$

$$\frac{\begin{array}{c} f :^{*} \tau_i \to \tau_0 \in \Gamma \\ \Gamma,\ f :^{*} \overrightarrow{\tau_i} \to \tau_0 \mid \Sigma \vdash s_1 : \tau \mid C \cup \{f\} \qquad \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \mid \Sigma \vdash s_2 : \tau \mid C \end{array}}{\Gamma \mid \Sigma \vdash\ \textbf{intercept}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\} : \tau \mid C}\ [\textsc{Intercept}]$$

$$\frac{\begin{array}{c} \Sigma(l) = \overrightarrow{\tau_i} \to \tau_0 \\ \Gamma \mid \Sigma \vdash s_1 : \tau \mid C \qquad \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \mid \Sigma \vdash s_2 : \tau \mid C \end{array}}{\Gamma \mid \Sigma \vdash\ \textbf{intercept}\ \{\textbf{cap}_l \Rightarrow s_1\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\} : \tau \mid C}\ [\textsc{CapIntercept}]$$

Figure 6.2: Extended Handler Intercept Typing Rules

Intercept. Similar to the Try rule, the Intercept rule type checks whether a handler statement is well-typed such that it produces the same output regardless if an effect is handled or not. In other words, we want to prevent a program being typed differently depending on whether an effect is triggered. Furthermore, to ensure that the interception of the handler is valid, the Intercept rule specifies that the handler which we are intercepting (i.e., f) must already exist in the environment ($f :^{*} \tau_i \to \tau_0 \in \Gamma$). Otherwise, we would be intercepting a non-existent capability f.

CapIntercept. The existence of the CapIntercept rule stems from the reduction rules presented in Figure 6.3. To evaluate the form

$$\textbf{intercept}\ \{f \Rightarrow s_1\}\,\textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\}$$

we replace the handler f with a corresponding location $\textbf{cap}_l$. Thus, after evaluation, we end up with

$$\textbf{intercept}\ \{\textbf{cap}_l \Rightarrow s_1\}\,\textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\}$$

As a result, it is prudent for us to also have an analogous typing rule which checks this reduced form of the interception. The only difference in this rule compared to Intercept is that we check the location l and l's corresponding type instead of a handler/capability f.

6.1.3 Reduction Rules

Finally, the reduction rule for the newly extended intercept handler construct is relatively simple. This is because we take advantage of the fact that the original System C language is already readily able to evaluate the results of handlers. This comes from the fact that intercepting a handler can be thought of as replacing one handler with another. Thus, the only thing which the intercept reduction rule has to do is reduce the program to a form which aligns with what would happen if a regular **try** block is evaluated.

$$\langle \textbf{intercept}\ \{\textbf{cap}_l \Rightarrow s\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi\rangle \longrightarrow \langle \#_l\{s\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi\rangle$$

Figure 6.3: Extended Reduction Rule for Intercept

6.2 Redex Modeling

6.2.1 Implementing the Model

Similar to Case Study 1, our model for the handler interception extension is a one-to-one translation from the design previously specified in this chapter. Therefore, we will not elaborate on much of the modeling process because there were no real complexities associated with the modeling. Figure 6.4, Figure 6.5, and Figure 6.6 display the Redex code used to model the syntax, typing rules, and reduction rules. The more interesting application of the Redex modeling comes from the safety errors that could be caught because we were able to run examples via our model. This is especially because this is what our approach contributes to the process of proving safety for a language. We expand upon this in the next section.

6.2.2 Running Examples and Preemptive Error Detection

As we stated previously, the more interesting discussion to be had about our Redex model for the handler interception is about the errors in safety that the model allowed us to discover. The first issue during our design process which was discovered was the syntax of our handler interception mechanism. Originally, the grammatical construct which we chose to represent our handler interception was:

$$\textbf{try } \{f \Rightarrow s\} \textbf{ intercept } \{f \Rightarrow s_1\} \textbf{with } \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\}$$

However, after running some examples using our Redex model, it became clear that this syntax was awkward. We questioned the purpose of using an intercept this way. If a developer is cognisant of the fact that an effect handler should be intercepted, they could just change the original handler instead of adding an intercepting handler.

Although this is not technically related to the safety of the language extension, making a language feature more ergonomic is of benefit to the longevity of a programming language. While it may be argued that this cumbersome design could have been discovered without a Redex model, we contend that running examples in a Redex model feels much closer to the experience of practically using a language. This is because, without a model, researchers would have to manually run through examples on paper. Dry run testing in this fashion may confirm that a language behaves as intended but does not give a feel for how actually programming in the language would be like.

Furthermore, our Redex model of the handler interception allowed us to detect an error in the type safety of our mechanism. After changing our syntax to be more ergonomic, our Intercept typing rule still did not check whether a handler statement produced the same output type regardless of whether an effect was handled or not. Let us use one of our examples to demonstrate. Take the program:

```
intercept {f ⇒ return 1} with { k ⇒ return True}
```

Although this program may seem correct at first glance, in order to type our intercept construct properly, the type of the output for this program must be consistent. In our case, we actually have two types which could be returned from this program depending on if our handler is called. If the handler is triggered, our program would return `True`. On the other hand, if the handler is not triggered, the program would return `1`. This is exactly the type of safety errors which we would expect our Redex model to detect.

However, Redex did not report the typing error in the example program which was run. Instead, the program was well-typed under our model. As such, we could tell that there must be something wrong with our typing rules and, after updating our typing rules, the program was no longer well-typed under our mode. This demonstrates another way in which Redex was able to help with detecting errors in a language's safety. Not only can a Redex model confirm the intended behaviour of a language by demonstrating that valid programs run correctly and can be type check, Redex modeling can also be used to confirm that invalid programs do indeed fail.

6.3 Safety

Similar to the first case study, the final step of our approach involves a proof of safety. Also, like the first case study, we defer the proof of safety for the extension implemented in this case study to Chapter 7. This is so that our proof of safety will consider the interactions between the extensions made in both of our case studies along with the original System C language. We must not that, as mentioned, our approach has already caught some errors before a proof of safety was even conducted. Thus, we can already state that our approach helped to enhance the efficacy of the programming languages research cycle in both of the case studies which we attempted.

6.4 Redex Code

See Figure 6.4, Figure 6.5, and Figure 6.6.

```
(s (def f = b #\; s)
   (b (e ... #\, b ...))
   (val x = s #\; s)
   (return e)
   (try f ⇒ s with h)
   (intercept f ⇒ s with h)
   (l s with h))
```

Figure 6.4: Interception Model Syntax

```
[(statement-type (g ... (f :* (τ_1 ... #\, → τ_0))) Σ s_1 τ (append f c) C)
 (statement-type (g ... (x_1 : τ_1) ... (k : C (τ_0 #\, → τ))) Σ s_2 τ c C)
 (where #t (subset C (append f c)))
 (where (τ_1 ... #\, → τ_0) (find-equal f (g ...)))
 ---------------------------------------------------------------- "Intercept"
 (statement-type (g ...) Σ (try f ⇒ s_1 with ((x_1 : τ_1) ... #\, (k : τ_0)
     ⇒ s_2)) τ c (set-minus (f) C))]

[(statement-type (g ...) Σ s_1 τ c C)
 (statement-type (g ... (x_1 : τ_1) ... (k : C (τ_0 #\, → τ))) Σ s_2 τ c C)
 (where (τ_1 ... #\, → τ_0) (find-equal l Σ))
 ------------------------------------------------------------- "CapIntercept"
 (statement-type (g ...) Σ (try (cap l) ⇒ s_1 with ((x_1 : τ_1) ... #\, (k
     : τ_0) ⇒ s_2)) τ c C)]
```

Figure 6.5: Interception Model Typing Rules

```
(--> (in-hole E (intercept (cap l) ⇒ s with ((x_1 : τ_1) ... #\, (k : τ_0)
    ⇒ s_prime)))
     (in-hole E (l with ((x_1 : τ_1) ... #\, (k : τ_0) ⇒ s_prime)))
     (fresh l)
     "interc")
```

Figure 6.6: Interception Model Reduction Rules

6.5 Complete Rules and Syntax

Finally, we present a completed typing rule and syntax with the addition of all contribution in Figure 6.8, Figure 6.9, and Figure 6.7.

Syntax:				
Expressions	e	$::=$	x	expression variables
		$\mid$	$() \mid 0 \mid 1 \mid ... \mid \text{true} \mid \text{false} \mid ...$	primitives
		$\mid$	$\textbf{box}\ b$	box introduction
		$\mid$	$\textbf{new}\ e$	initialisation
		$\mid$	$!e$	dereferencing
		$\mid$	$e \coloneqq e$	assignment
		$\mid$	l	store location
Blocks	b	$::=$	f	block variables
		$\mid$	$\left\{\left(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j} \Rightarrow s\right)\right\}$	block implementation
		$\mid$	$\textbf{unbox}\ e$	box elimination
Statements	s	$::=$	$\textbf{def}\ f = b;\ s$	block definition
		$\mid$	$b\left(\overrightarrow{e_i}, \overrightarrow{b_j}\right)$	block application
		$\mid$	$\textbf{val}\ x = s;\ s$	sequencing
		$\mid$	$\textbf{return}\ e$	returning
		$\mid$	$\textbf{try}\ \{f \Rightarrow s\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s\right\}$	handlers
		$\mid$	$\textbf{intercept}\ \{f \Rightarrow s\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s\right\}$	handlers interception
Types:				
Value Types	τ	$::=$	$\text{Int} \mid \text{Boolean} \mid ...$	base types
		$\mid$	$\sigma\ \textbf{at}\ C$	boxed block types
		$\mid$	$\textbf{Ref}\ \tau$	reference types
Block Types	σ	$::=$	$\left(\overrightarrow{\tau_i}, \overrightarrow{f_j : \sigma_j}\right) \to \tau$	
Capabilities	C	$::=$	$\varnothing \mid \{f\} \mid C \cup C$	
Environments:				
Environments	Γ	$::=$	$\varnothing$	empty environment
		$\mid$	$\Gamma, x : \tau$	value bindings
		$\mid$	$\Gamma, f :^{*} \sigma$	tracked bindings
		$\mid$	$\Gamma, f :^{C} \sigma$	transparent bindings

Figure 6.7: Syntax of the Extended System C Language

Block Typing. $\boxed{\Gamma \mid \Sigma \vdash b : \sigma \mid C}$

$$\frac{f :^{C} \sigma \in \Gamma}{\Gamma \mid \Sigma \vdash f : \sigma \mid C}\ [\textsc{Transparent}] \qquad \frac{f :^{*} \sigma \in \Gamma}{\Gamma \mid \Sigma \vdash f : \sigma \mid \{f\}}\ [\textsc{Tracked}]$$

$$\frac{\Gamma,\ \overrightarrow{x_i : \tau_i},\ \overrightarrow{g_j :^{*} \sigma_j} \mid \Sigma \vdash s : \tau \mid C \cup \overrightarrow{g_j}}{\Gamma \mid \Sigma \vdash \{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j : \sigma_j}) \Rightarrow s\} : (\overrightarrow{\tau_i}, \overrightarrow{g_j : \sigma_j}) \to \tau \mid C}\ [\textsc{Block}]$$

$$\frac{\Gamma \mid \Sigma \vdash b : \sigma \mid C'}{\Gamma \mid \Sigma \vdash \textbf{unbox}\ e : \sigma \mid C}\ [\textsc{BoxElim}] \qquad \frac{\Gamma \mid \Sigma \vdash e : \sigma\ \textbf{at}\ C \qquad C' \subseteq C}{\Gamma \mid \Sigma \vdash b : \sigma \mid C}\ [\textsc{BSub}]$$

Expression Typing. $\boxed{\Gamma \mid \Sigma \vdash e : \tau}$

$$\frac{}{\Gamma \mid \Sigma \vdash n : \mathrm{Int}}\ [\textsc{Lit}] \qquad \frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash x : \tau}\ [\textsc{Var}]$$

$$\frac{\Gamma \mid \Sigma \vdash b : \sigma \mid C}{\Gamma \mid \Sigma \vdash \textbf{box}\ b : \sigma\ \textbf{at}\ C}\ [\textsc{BoxIntro}] \qquad \frac{\Gamma \mid \Sigma \vdash e : \textbf{Ref}\ \tau}{\Gamma \mid \Sigma \vdash !e : \tau}\ [\textsc{Deref}]$$

$$\frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \textbf{new}\ e : \textbf{Ref}\ \tau}\ [\textsc{New}] \qquad \frac{\Gamma \mid \Sigma \vdash e_1 : \textbf{Ref}\ \tau \qquad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1 \coloneqq e_2 : \tau}\ [\textsc{Assign}] \qquad \frac{\Sigma(l) = \tau}{\Gamma \mid \Sigma \vdash l : \textbf{Ref}\ \tau}\ [\textsc{Ref}]$$

Statement Typing. $\boxed{\Gamma \mid \Sigma \vdash s : \tau \mid C}$

$$\frac{\Gamma \mid \Sigma \vdash s_0 : \tau_0 \mid C_0 \qquad \Gamma,\ x : \tau_0 \mid \Sigma \vdash s_1 : \tau_1 \mid C_1}{\Gamma \mid \Sigma \vdash \textbf{val}\ x = s_0;\ s_1 : \tau_1 \mid C_0 \cup C_1}\ [\textsc{Val}] \qquad \frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \textbf{return}\ e : \tau \mid \varnothing}\ [\textsc{Ret}]$$

$$\frac{\Gamma \mid \Sigma \vdash b : (\overrightarrow{\tau_i}, \overrightarrow{f_j : \sigma_j}) \to \tau \mid C \qquad \overrightarrow{\Gamma \mid \Sigma \vdash e_i : \tau_i} \qquad \overrightarrow{\Gamma \mid \Sigma \vdash b_j : \sigma_j \mid C_j}}{\Gamma \mid \Sigma \vdash b(\overrightarrow{e_i}, \overrightarrow{b_j}) : \tau[\overrightarrow{f_j \mapsto C_j}] \mid C \cup \overrightarrow{C_j}}\ [\textsc{App}]$$

$$\frac{\Gamma \mid \Sigma \vdash b : \sigma \mid C' \qquad \Gamma, f :^{C'} \sigma \mid \Sigma \vdash s : \tau \mid C}{\Gamma \mid \Sigma \vdash \textbf{def}\ f = b;\ s : \tau \mid C}\ [\textsc{Def}] \qquad \frac{\Gamma \mid \Sigma \vdash s : \tau \mid C' \qquad C' \subseteq C}{\Gamma \mid \Sigma \vdash s : \tau \mid C}\ [\textsc{SSub}]$$

$$\frac{\Gamma,\ f :^{*} \overrightarrow{\tau_i} \to \tau_0 \mid \Sigma \vdash s_1 : \tau \mid C \cup \{f\} \qquad \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \mid \Sigma \vdash s_2 : \tau \mid C}{\Gamma \mid \Sigma \vdash \textbf{try}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} : \tau \mid C}\ [\textsc{Try}]$$

$$\frac{\begin{array}{c} f :^{*} \tau_i \to \tau_0 \in \Gamma \\ \Gamma,\ f :^{*} \overrightarrow{\tau_i} \to \tau_0 \mid \Sigma \vdash s_1 : \tau \mid C \cup \{f\} \qquad \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \mid \Sigma \vdash s_2 : \tau \mid C \end{array}}{\Gamma \mid \Sigma \vdash\ \textbf{intercept}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\} : \tau \mid C}\ [\textsc{Intercept}]$$

$$\frac{\begin{array}{c} \Sigma(l) = \overrightarrow{\tau_i} \to \tau_0 \\ \Gamma \mid \Sigma \vdash s_1 : \tau \mid C \qquad \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \mid \Sigma \vdash s_2 : \tau \mid C \end{array}}{\Gamma \mid \Sigma \vdash\ \textbf{intercept}\ \{\textbf{cap}_l \Rightarrow s_1\}\ \textbf{with}\ \left\{\left(\overrightarrow{x_i}, k\right) \Rightarrow s_2\right\} : \tau \mid C}\ [\textsc{CapIntercept}]$$

Figure 6.8: Language Static Semantics

Evaluation Contexts:

$$
\begin{array}{lll}
\text{Contexts} \;\; \mathrm{E} & ::= & \square \mid \mathbf{val}\ x = \mathrm{E};\ s \mid b(\overrightarrow{E_i}, \overrightarrow{b_j}) \mid \#\{\mathrm{E}\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s\} \\
\text{Contexts} \;\; \mathrm{H}_l & \coloneqq & \square \mid \mathbf{val}\ x = \mathrm{H}_l;\ s \mid b(\overrightarrow{H_i}, \overrightarrow{b_j}) \mid \#_{l'}\{\mathrm{H}_l\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s\} \quad \text{where } l \neq l'
\end{array}
$$

Values:

$$
\begin{array}{lll}
\text{Expression Values} \;\; v & ::= & () \mid 0 \mid 1 \mid \ldots \mid \mathrm{true} \mid \mathrm{false} \mid \ldots \mid \mathbf{box}\ w \mid l \\
\text{Block Values} \;\; w & ::= & \{(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s\} \mid \mathbf{cap}_l
\end{array}
$$

Reduction Rules:

$$
\begin{array}{llll}
(\mathit{box}) & \langle \mathbf{unbox}\ (\mathbf{box}\ b) \mid \Xi \rangle & \longrightarrow & \langle b \mid \Xi \rangle \\
(\mathit{val}) & \langle \mathbf{val}\ x = \mathbf{return}\ v;\ s \mid \Xi \rangle & \longrightarrow & \langle s[x \mapsto v] \mid \Xi \rangle \\
(\mathit{def}) & \langle \mathbf{def}\ f = w;\ s \mid \Xi \rangle & \longrightarrow & \langle s[f \mapsto w] \mid \Xi \rangle \\
(\mathit{ret}) & \langle \#_l\{\ \mathbf{return}\ v\}\ \ \mathbf{with}\ h \mid \Xi \rangle & \longrightarrow & \langle v \mid \Xi \rangle \\
(\mathit{app}) & \langle (\{(\overrightarrow{x_i}, \overrightarrow{f_j}) \Rightarrow s\})(\overrightarrow{v_i}, \overrightarrow{w_j}) \mid \Xi \rangle & \longrightarrow & \langle s[\overrightarrow{x_i \mapsto v_i}, \overrightarrow{f_j \mapsto C_j}, \overrightarrow{f_j \mapsto w_j}] \mid \Xi \rangle \\
 & \quad \text{where } \varnothing \vdash w_j : \sigma_j \mid C_j & & \\
(\mathit{try}) & \langle \mathbf{try}\ \{f \Rightarrow s\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi \rangle & \longrightarrow & \langle \#_l\{s[f \mapsto \{l\}, f \mapsto \mathbf{cap}_l]\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi \rangle \\
 & \quad \text{where } l \notin \mathit{dom}\ \Xi, \text{ and } \vdash f : \overrightarrow{\tau_i} \to \tau_0, \text{ then } \Xi(l) \coloneqq \overrightarrow{\tau_i} \to \tau_0 & & \\
(\mathit{interc}) & \langle \mathbf{intercept}\ \{\mathbf{cap}_l \Rightarrow s\}\ \mathbf{with} & \longrightarrow & \langle \#_l\{s\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi \rangle \\
 & \quad \{(\overrightarrow{x_i}, k) \Rightarrow s'\} \mid \Xi \rangle & & \\
(\mathit{cap}) & \langle \#_l\{H_l[\mathbf{cap}_l(\overrightarrow{v_i})]\}\ \mathbf{with}\ h \mid \Xi \rangle & \longrightarrow & \langle s[\overrightarrow{x_i \mapsto v_i}, k \mapsto \{y \Rightarrow \#_l\{H_l[\mathbf{return}\ y]\}\ \mathbf{with}\ h\}] \mid \Xi \rangle \\
 & \quad \text{where } h = \{(\overrightarrow{x_i}, k) \Rightarrow s\} & & \\
(\mathit{deref}) & \langle !l \mid \Xi \rangle & \longrightarrow & \langle v \mid \Xi \rangle \\
 & \quad \text{where } \Xi(l) = v & & \\
(\mathit{assign}) & \langle l \coloneqq v \mid \Xi \rangle & \longrightarrow & \langle v \mid \Xi[l \mapsto v] \rangle \\
(\mathit{new}) & \langle \mathbf{new}\ v \mid \Xi \rangle & \longrightarrow & \langle l \mid \Xi,\ l \mapsto v \rangle \\
 & \quad \text{where } l \notin \mathit{dom}\ \Xi & &
\end{array}
$$

Figure 6.9: Extended System C Reduction Rules

Chapter 7

Safety

Having completed our language extensions for System C in the two case studies which we attempted, we now attempt a proof of safety. Given that we already modeled the extensions to System C in Redex according to our approach, we should not encounter any major errors in safety while attempting this proof. One thing that we must note is that there is already an outline of a safety proof for System C in Brachthäuser et al. (2022)'s paper. However, the safety proof presented does not go into much detail. Thus, we elect to not skip any of the details of our safety proof and provide a more comprehensive proof of safety for System C in addition to our case study extensions. The following is a proof of progress and preservation.

7.1 Progress

Progress. If s is a closed, well-typed term with an empty context and Ξ is a well-typed store under Γ, s will not become stuck. That is to say, s must be a value or $\langle s \mid \Xi \rangle \longrightarrow \langle s' \mid \Xi' \rangle$ for some s' and Ξ'.

We also prove mutual induction on expressions and blocks. If e is a closed, well-typed term with an empty context and Ξ is a well-typed store under Γ, e will not become stuck. That is to say, e must be an expression value v or $\langle e \mid \Xi \rangle \longrightarrow \langle e' \mid \Xi' \rangle$ for some e' and Ξ'.

Furthermore, if b is a closed, well-typed term with an empty context and Ξ is a well-typed store under Γ, b will not become stuck. That is to say, b must be an expression value v or $\langle b \mid \Xi \rangle \longrightarrow \langle b' \mid \Xi' \rangle$ for some b' and Ξ'.

Theorem 1 (Progress of Statements)**.** *If* $\langle\, \varnothing \mid \Sigma \vdash s : \tau \mid \varnothing\, \Xi \,\rangle$*, then* s *is* ***return*** v *or* $\langle s \mid \Xi \rangle \longmapsto \langle s' \mid \Xi' \rangle$*.*
Proof: *We first present a induction on a derivation of* $s : \tau$ *with cases over the many*

forms of s.

Case Val**:** $\langle \textbf{val}\ x = s_1;\ s_2 \mid \Xi\rangle$
According to the induction hypothesis, s_1 is either of the form **return** v or can be reduced to a further term.

- If s_1 is **return** v, by using the *(val)* reduction rule, s is able to be reduced further to $s_2[x \mapsto v]$. Thus:

$$\langle \textbf{val}\ x = s_1;\ s_2 \mid \Xi\rangle \longmapsto \langle s_2[x \mapsto v] \mid \Xi\rangle$$

- If s_1 is not **return** v, according to the induction hypothesis, it will eventually become a **return** v. As such, the *(cong)* reduction rule in conjunction with the induction hypothesis guarantees that the form of s_1 will always eventually be **return** v.

Case Ret**:** $\langle \textbf{return}\ e \mid \Xi\rangle$
Let s be **return** e. In this situation, there are two sub-cases for e.

- If e is an expression variable x, s is not well-formed. This is because the Var rule would not be valid because the variable x would not exist in the empty environment Γ.
- If e is a primitive or box introduction, this case is trivially true because s would be of the form **return** v.

Case App**:** $\langle b(\overrightarrow{e_i}, \overrightarrow{b_j}) \mid \Xi\rangle$
Let s be $b(\overrightarrow{e_i}, \overrightarrow{b_j})$. In this situation, there are 3 sub-cases which we need to consider.

- If b is of the form $\left\{\left(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j} \Rightarrow s\right)\right\}$, e_i is of the form v_i and b_j is of the form w_j, the *(app)* rule allows for evaluation of s such that:

$$\langle(\{(\overrightarrow{x_i}, \overrightarrow{f_j}) \Rightarrow s\})(\overrightarrow{v_i}, \overrightarrow{w_j}) \mid \Xi\rangle \longrightarrow \langle s[\overrightarrow{x_i \mapsto v_i}, \overrightarrow{f_j \mapsto C_j}, \overrightarrow{f_j \mapsto w_j}] \mid \Xi\rangle$$

- If b were of the form **unbox** (**box** b), s would not be well-typed. Thus, we do not need to consider this case.
- If b were of the form f, this would invalidate the Transparent rule because the f block variable would not be in the environment Γ.

Case Def**:** $\langle \textbf{def}\ f = b;\ s \mid \Xi\rangle$
Let s be **def** $f = b;\ s$. In this situation, there are 3 sub-cases.

- If b is of the form f, s would not be well-typed. This is because of the Transparent rule and the fact that the capability f would not exist in the environment Γ. As such, the $f :^{C} \sigma \in \Gamma$ portion of the Transparent typing rule would be violated.

- If b is of the form $\left\{\left(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j} \Rightarrow s\right)\right\}$, the reduction rule *(def)* applies such that:

$$\langle \mathbf{def}\ f = b;\ s_1 \mid \Xi \rangle \longmapsto \langle s_1[f \mapsto b] \mid \Xi \rangle$$

- If b is of the form **unbox** e, s would not be well-typed. Thus, we do not need to consider this case.

Case SSub**:** $\langle s \mid \Xi \rangle$
Let s be s. This case is trivially true because of the induction hypothesis.

Case Try**:** $\langle \mathbf{try}\ \{f \Rightarrow s_1\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} \mid \Xi \rangle$
Let s be **try** $\{f \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. By applying the *(try)* reduction rule, s always reduces, regardless of what form s takes, such that:

$$\begin{aligned} &\langle \mathbf{try}\ \{f \Rightarrow s_1\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} \mid \Xi \rangle \\ \longmapsto &\langle \#_l\{s_1[f \mapsto \{l\}, f \mapsto \mathbf{cap}_l]\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} \mid \Xi' \rangle \end{aligned}$$

In this case, the Ξ' would be Ξ but with the inclusion of a fresh location l as according to the **try** reduction rule. Thus, the property of progress holds.

Case Cap**:** $\langle \mathbf{cap}_l \mid \Xi \rangle$
Before we prove this case, we must first prove a lemma.

Lemma 1 (Delimited Labels)**.** *We will lift this lemma directly from the Brachthäuser et al. (2022) paper because it is one of the only parts of the safety proof which is actually explicitly stated by the authors. The lemma states that if* $\varnothing \vdash E[\boldsymbol{cap}_l(\overrightarrow{v_i})] : \tau \mid \varnothing$ *and* $\Gamma \vdash \boldsymbol{cap}_l(\overrightarrow{v_i}) : \tau' \mid C$ *and* $\vdash_{ctx} E : \tau' \rightsquigarrow \tau \mid C$*, then* $E = E'[\#_l\{H_l\}\ \boldsymbol{with}\ h]$*.*

To continue on with the proof, we can split s into an evaluation context and redex. This is immediately obvious because for a $\mathbf{cap}_l$ to occur, the term would have to have been evaluated from within a **try** block. Thus, if we consider that $s = E[s_{redex}]$, s_{redex} can take on a few forms:

- If s_{redex} is of the form $\mathbf{cap}_l(\overrightarrow{v_i})$, we can use the *(cong)* and *(cap)* rules considering that the s must be within a suitable evaluation context.
- If s_{redex} is of any other form, we can use the delimited labels lemma alongside the *(cap)* rule. This would ensure that there is a step able to be taken according to the *(cap)* rule.

Case Reset**:** $\langle \#_l\{s_1\}\ \mathbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} \mid \Xi \rangle$
When $s :\Rightarrow \#_l\{s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$, s_1 can take multiple forms:

- If s_1 is of the form **return** v, we can directly take a step according to the *(ret)* reduction rule. We note that if s_1 is of the form **return** x, this would not be well-typed and thus, we do not need to consider this case.
- If s_1 is of any other form, according to the *(cong)* rule, the evaluation context

$\#_l\{E\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s\}$ and the induction hypothesis, s_1 will eventually reach the form of **return** v. Thus, s will be able to take a step according to the *(ret)* rule.

Case Intercept**:** $\langle$**intercept** $\{f \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\} \mid \Xi\rangle$
We do not need to consider this case because it does not type check under an empty Γ. As such, even as we build our proof inductively, there will be no case where we need to consider a non-empty Γ.

Case CapIntercept**:** $\langle$**intercept** $\{\mathbf{cap}_l \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\} \mid \Xi\rangle$
When s is of the form **intercept** $\{\mathbf{cap}_l \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$, we can apply the *(interc)* evaluation rule regardless of the specific form of s_1 or s_2. Thus, s can take a step and the property of progress holds.

Proof: If $\varnothing \mid \Sigma \vdash b : \sigma \mid C$, then b is a block value w or $\langle b \mid \Xi\rangle \longmapsto \langle b' \mid \Xi'\rangle$. A block value w is defined as anything of the form $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s\} \mid \mathbf{cap}_l$. We first present a induction on a derivation of $b : \tau$ with cases over the many forms of b.

Case Transparent **and** Tracked**:** $\langle f \mid \Xi\rangle$ and $\langle f \mid \Xi\rangle$
We do not need to consider these cases because they do not type check under an empty Γ. As such, even as we build our proof inductively, there will be no case where we need to consider a non-empty Γ.

Case Block**:** $\langle\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j :^* \sigma_j}) \Rightarrow s\} \mid \Xi\rangle$
When b is of the form $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j :^* \sigma_j}) \Rightarrow s\}$, the case is trivially true because b is a block value w.

Case BoxElim**:** $\langle$**unbox** $e \mid \Xi\rangle$
When b is of the form **unbox** e, e must be of the form **box** b_1 for some arbitrary b_1 else b would be ill-typed. Thus, for b of the form **unbox** (**box** b_1), we can apply the *(box)* reduction rule to let b take a step.

Case BSub**:** $\langle b \mid \Xi\rangle$
This case is trivially true because of the induction hypothesis.

Proof: If $\varnothing \mid \Sigma \vdash e : \sigma$, then e is an expression value v or $\langle e \mid \Xi\rangle \longmapsto \langle e' \mid \Xi'\rangle$. An expression value v is defined as anything of the form $() \mid 0 \mid 1 \mid \ldots \mid \text{true} \mid \text{false} \mid \ldots \mid$ **box** $w \mid l$. We first present a induction on a derivation of $e : \tau$ with cases over the many forms of e.

Case Lit **and** BoxIntro**:** $\langle n \mid \Xi\rangle$ and $\langle x \mid \Xi\rangle$
Trivially true because e can not step to anything.

Case Deref**:** $\langle !e \mid \Xi\rangle$
When $e :\Rightarrow !e$, e can take on multiple forms:

- If e of the form l, we can apply the *(deref)* reduction rule. Thus, the program will be able to step and the progress theorem holds.
- If e is of any other form, e would no longer be well-typed or valid. Thus, these sub-cases are trivial.

Case New**:** $\langle \textbf{new } e \mid \Xi \rangle$
When $e :\Rightarrow \textbf{new } e$, e can take on multiple forms:

- If e is of the form v excluding l, the *(new)* reduction rule applies and e can take a step. Thus, the induction hypothesis holds.
- If e is of any other form, e would not be well-typed or valid. Thus, we do not need to consider these sub-cases.

Case Assign**:** $\langle e_1 \coloneqq e_2 \mid \Xi \rangle$
When $e :\Rightarrow e_1 \coloneqq e_2$, e_1 and e_2 could take on multiple forms:

- If e_1 is of the form l and e_2 is of the form v excluding l, we can apply the *(assign)* reduction rule. Thus, e is able to step and the induction hypothesis holds.
- If e_1 and e_2 are of any other form, e_1 and e_2 would not be well-typed or valid. Thus, we do not need to consider these sub-cases.

Case Ref**:** $\langle l \mid \Xi \rangle$
Trivial because this case requires that Σ be non-empty for it to be well-typed. However, because our induction hypothesis is under an empty Σ, we do not need to consider this case and we will never build any non-empty Σ.

Case Var**:** $\langle x \mid \Xi \rangle$
We do not need to consider this case because it does not type check under an empty Γ. As such, even as we build our proof inductively, there will be no case where we need to consider a non-empty Γ.

7.2 Preservation

Preservation. For a term s, e or b. If the term is well-typed, then the type of the reduced term s', e' or b' should be preserved from the original term. That is to say, the type of a term should remain the same after evaluation for valid terms.

7.2.1 Lemmas

Before we prove the property of preservation, we first need to establish a prove a few lemmas which will help with the overall proof.

Lemma 2 (Substitution of expressions)**.** *If* $\Gamma, e_1 : \tau' \mid \Sigma \vdash s : \tau \mid C$ *and* $\Gamma \mid \Sigma \vdash e_2 : \tau' \mid C$*, then* $\Gamma \mid \Sigma \vdash s[e_1 \mapsto e_2] : \tau \mid C$.

Proof. By mutual induction on the typing of the statement $\Gamma, e_1 : \tau' \mid \Sigma \vdash s : \tau \mid C$, $\Gamma, e_1 : \tau' \mid \Sigma \vdash b : \sigma \mid C$, and $\Gamma, e_1 : \tau' \mid \Sigma \vdash e : \tau$. We do an induction on the cases of the final typing rules which are able to be used.

Case Val:
For this case, let s be **val** $x = s_0; s_1$. In this situation there are two sub-cases:

- When $e_1 = x$, $s[e_1 \mapsto e_2]$ is equivalent to **val** $x[x \mapsto e_2] = s_0[x \mapsto e_2]; s_1[x \mapsto e_2]$. Thus, we get **val** $e_2 = s_0[x \mapsto e_2]; s_1[x \mapsto e_2]$ after substitution. From the induction hypothesis, we know that s_0 and s_1 maintain their types after substitution. Therefore, we can say that the overall type is maintained after substitution.
- Otherwise, $s[e_1 \mapsto e_2]$ is equivalent to **val** $x = s_0[e_1 \mapsto e_2]; s_1[e_1 \mapsto e_2]$. Thus, by the induction hypothesis, we can say that s_0 and s_1 maintain their types even after substitution. Therefore, we can say that the overall type is maintained after substitution.

Case Ret:
For this case, let s be **return** e. In this situation, there are two sub-cases:

- When $e_1 = e$, $s[e_1 \mapsto e_2]$ is equivalent to **return** $e[e \mapsto e_2]$. Thus, we get **return** e_2 after this substitution is completed. Because $e : \tau'$ (from the fact that $e_1 = e$) and $e_2 : \tau'$, we know that substitution must preserve types. Thus, the lemma is satisfied.
- If $e_1 \neq e$, $s[e_1 \mapsto e_2]$ is equivalent to **return** $e[e_1 \mapsto e_2]$. Thus, we get **return** e after this substitution is completed. This trivially satisfies the lemma because nothing was substituted.

Case App:
For this case, let s be $b(\overrightarrow{e_i}, \overrightarrow{b_j})$. In this situation, there are two sub-cases:

- If $e_1 \in \overrightarrow{e_i}$, we can replace all instances of e_1 in $\overrightarrow{e_i}$ with e_2. However, this should not change the type of any expression in $\overrightarrow{e_i}$ because $e_1 : \tau'$ and $e_2 : \tau'$. Then, we can say that $s[e_1 \mapsto e_2]$ is equivalent to $b[e_1 \mapsto e_2](\overrightarrow{e_i}[e_1 \mapsto e_2], \overrightarrow{b_j}[e_1 \mapsto e_2])$. From the mutual induction hypothesis, we know that type is preserved after substitution in blocks. Thus, from the App typing rule, we can say that $b[e_1 \mapsto e_2](\overrightarrow{e_i}[e_1 \mapsto e_2], \overrightarrow{b_j}[e_1 \mapsto e_2]) : \tau$. Thus, the lemma is satisfied.
- If $e_1 \notin \overrightarrow{e_i}$, $s[e_1 \mapsto e_2]$ is equivalent to $b[e_1 \mapsto e_2](\overrightarrow{e_i}, \overrightarrow{b_j}[e_1 \mapsto e_2])$. From the mutual induction hypothesis, we know that type is preserved after substitution in blocks. Thus, from the App typing rule, we can say that $b[e_1 \mapsto e_2](\overrightarrow{e_i}, \overrightarrow{b_j}[e_1 \mapsto e_2]) : \tau$. Thus, the lemma is satisfied.

Case Def:
For this case, we let s be **def** $f = b;\ s'$. In this case, we can say that, after substitution, s would be **def** $f = b[e_1 \mapsto e_2];\ s'[e_1 \mapsto e_2]$. From the mutual induction hypothesis, we know that b and s' maintain their types after substitution. Thus, from the Def typing

rule, we can say that the type of the overall term s has been preserved after substitution. Thus, the lemma is satisfied.

Case SSub**:**
For this case, let s be s. Then, this case is trivially true because of the induction hypothesis.

Case Try**:**
For this case, we let s be $\textbf{try}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. After substitution, s would be $\textbf{try}\ \{f \Rightarrow s_1[e_1 \mapsto e_2]\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k[e_1 \mapsto e_2]) \Rightarrow s_2[e_1 \mapsto e_2]\}$. From the induction hypothesis, we know that s_1 and s_2 maintain their types after substitution. Furthermore, k can be considered a block, thus we can also apply the induction hypothesis. Thus, from the Try rule, we can say that the type of the overall term s has been preserved after substitution. Thus, the lemma is satisfied.

Case Intercept**:**
For this case, we let s be $\textbf{intercept}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. After substitution, s would be $\textbf{intercept}\ \{f \Rightarrow s_1[e_1 \mapsto e_2]\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k[e_1 \mapsto e_2]) \Rightarrow s_2[e_1 \mapsto e_2]\}$. From the induction hypothesis, we know that s_1 and s_2 maintain their types after substitution. Furthermore, k can be considered a block, thus we can also apply the induction hypothesis. Thus, from the Intercept rule, we can say that the type of the overall term s has been preserved after substitution. Thus, the lemma is satisfied.

Case CapIntercept**:**
For this case, we let s be $\textbf{intercept}\ \{\textbf{cap}_l \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. After substitution, s would be $\textbf{intercept}\ \{\textbf{cap}_l \Rightarrow s_1[e_1 \mapsto e_2]\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k[e_1 \mapsto e_2]) \Rightarrow s_2[e_1 \mapsto e_2]\}$. From the induction hypothesis, we know that s_1 and s_2 maintain their types after substitution. Furthermore, k can be considered a block, thus we can also apply the induction hypothesis. Thus, from the CapIntercept rule, we can say that the type of the overall term s has been preserved after substitution. Thus, the lemma is satisfied.

Proof. The following are for the mutual induction cases on blocks where we prove that if $\Gamma, e_1 : \tau' \mid \Sigma \vdash b : \sigma \mid C$ and $\Gamma \mid \Sigma \vdash e_2 : \tau' \mid C$, then $\Gamma \mid \Sigma \vdash b[e_1 \mapsto e_2] : \sigma \mid C$.

Case Transparent **and** Tracked **:**
For this case, we let b be f. Thus, these cases are trivially true because there is nothing to be substituted.

Case Block**:**
For this case, we let b be $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j : \sigma_j}) \Rightarrow s'\}$. Therefore, after substitution, b would be $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j[e_1 \mapsto e_2] : \sigma_j}) \Rightarrow s'[e_1 \mapsto e_2]\}$. From the induction hypothesis, we know that blocks $(\overrightarrow{g_j})$ and s' maintain their types after substitution. Thus, from the Block rule, we can say that the type of the overall term b has been preserved after substitution. Thus, the lemma is satisfied.

Case BoxElim**:**

For this case, we let b be **unbox** e. In this situation, we have two sub-cases:

- If $e_1 = e$, $b[e_1 \mapsto e_2]$ is equivalent to **unbox** $e[e \mapsto e_2]$. Thus, we get **unbox** e_2 after substitution. We know that e and e_2 share the same type because $e_1 = e$. Thus, by applying the BoxElim rule, we can say that b preserves type even after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq e$, there is no substitution to be done. Thus, this case is immediately obvious and trivial.

Case BSub**:**
For this case, we let b be b. Thus, this is trivially true because of the induction hypothesis.

Proof. The following are for the mutual induction cases on blocks where we prove that if $\Gamma, e_1 : \tau \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash e_2 : \tau$, then $\Gamma \mid \Sigma \vdash e[e_1 \mapsto e_2] : \tau$.

Case Lit**:**
For this case, we let e be n. In this situation there are two sub-cases:

- If $e_1 = n$, e would be $n[n \mapsto e_2]$. Thus, after substitution, e would be e_2. We know that e_2 maintains the type of n because $e_1 : \tau = n : \tau$. Thus, from the Lit rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq n$, e would be $n[e_1 \mapsto e_2]$. Thus, after substitution, e would be n. Thus, the desired result is immediately obvious because there is no substitution. Thus, the lemma is satisfied.

Case Var**:**
For this case, we let e be x. In this situation there are two sub-cases:

- If $e_1 = x$, e would be $x[x \mapsto e_2]$. Thus, after substitution, e would be e_2. We know that e_2 maintains the type of x because $e_1 : \tau = x : \tau$. Thus, from the Var rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq x$, e would be $x[e_1 \mapsto e_2]$. Thus, after substitution, e would be x. Thus, the desired result is immediately obvious because there is no substitution. Thus, the lemma is satisfied.

Case BoxIntro**:**
For this case, we let e be **box** b. After substitution, e would be **box** $b[e_1 \mapsto e_2]$. From the mutual induction hypothesis, we know that b preserves its type after substitution. Thus, we can build a typing relation from the BoxIntro rule which tells us that the type of the overall term e remains the same after substitution. Thus, the lemma is satisfied.

Case Deref**:**
For this case, we let e be $!e$. In this situation there are two sub-cases:

- If $e_1 = e$, e would be $!e[e \mapsto e_2]$. Thus, after substitution, e would be $!e_2$. We know that e_2 maintains the type of e because $e_1 : \tau = e : \tau$. Thus, from the Deref rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq e$, e would be $!e[e_1 \mapsto e_2]$. Thus, after substitution, e would be $!e$. Thus, the desired result is immediately obvious because there is no substitution. Thus, the lemma is satisfied.

Case New**:**
For this case, we let e be **new** e. In this situation there are two sub-cases:

- If $e_1 = e$, e would be **new** $e[e \mapsto e_2]$. Thus, after substitution, e would be **new** e_2. We know that e_2 maintains the type of e because $e_1 : \tau = e : \tau$. Thus, from the New rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq e$, e would be **new** $e[e_1 \mapsto e_2]$. Thus, after substitution, e would be **new** e. Thus, the desired result is immediately obvious because there is no substitution. Thus, the lemma is satisfied.

Case Assign**:**
For this case, we let e be $e_3 \coloneqq e_4$. In this situation there are four sub-cases:

- If $e_1 = e_3$, e would be $e_3[e_3 \mapsto e_2] \coloneqq e_4[e_3 \mapsto e_2]$. Thus, after substitution, e would be $e_2 \coloneqq e_4$. We know that e_2 maintains the type of e_3 because $e_1 : \tau = e_3 : \tau$. Thus, from the Assign rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 = e_4$, e would be $e_3[e_4 \mapsto e_2] \coloneqq e_4[e_4 \mapsto e_2]$. Thus, after substitution, e would be $e_3 \coloneqq e_2$. We know that e_2 maintains the type of e_4 because $e_1 : \tau = e_4 : \tau$. Thus, from the Assign rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 = e_3 = e_4$, e would be $e_3[e_3 \mapsto e_2] \coloneqq e_4[e_4 \mapsto e_2]$. Thus, after substitution, e would be $e_2 \coloneqq e_2$. We know that e_2 maintains the type of e_3 and e_4 because $e_1 : \tau = e_3 : \tau = e_4 : \tau$. Thus, from the Assign rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq e_3$ and $e_1 \neq e_4$, e would be $e_3[e_1 \mapsto e_2] \coloneqq e_4[e_1 \mapsto e_2]$. Thus, after substitu-

tion, e would be $e_3 \coloneqq e_4$. Thus, the desired result is immediately obvious because there is no substitution. Thus, the lemma is satisfied.

Case Ref**:**
For this case, we let e be l. In this situation there are two sub-cases:

- If $e_1 = l$, e would be $l[l \mapsto e_2]$. Thus, after substitution, e would be l. We know that e_2 maintains the type of l because $e_1 : \tau = l : \tau$. Thus, from the Ref rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.
- If $e_1 \neq e$, e would be $l[e_1 \mapsto e_2]$. Thus, after substitution, e would be l. Thus, the desired result is immediately obvious because there is no substitution. Thus, the lemma is satisfied.

Lemma 3 (Substitution of blocks)**.** *If* $\Gamma, b_1 : \sigma \mid \Sigma \vdash s : \tau \mid C$ *and* $\Gamma \mid \Sigma \vdash b_2 : \sigma \mid C$, *then* $\Gamma \mid \Sigma \vdash s[b_1 \mapsto b_2] : \tau \mid C$.

Proof. By mutual induction on the typing of the statement $\Gamma, b_1 : \sigma \mid \Sigma \vdash s : \tau \mid C$, $\Gamma, b_1 : \sigma \mid \Sigma \vdash b : \sigma \mid C$ and $\Gamma, b_1 : \sigma \mid \Sigma \vdash e : \tau$. We do an induction on the cases of the final typing rules which are able to be used.

Case Val**:**
For this case, we let s be **val** $x = s_0; s_1$. After substitution, s would be **val** $x = s_0[b_1 \mapsto b_2]; s_1[b_1 \mapsto b_2]$. From the induction hypothesis, we know that s_0 and s_1 maintain their types after substitution. Thus, from the Val rule, we can construct the typing relation for the substituted s and say that the type of the overall term s has been preserved after substitution. Thus, the lemma is satisfied.

Case Ret**:**
For this case, we let s be **return** e. After substitution, s would be **return** $e[b_1 \mapsto b_2]$. From the mutual induction hypothesis, we know that e maintains it type after substitution. Thus, from the Ret rule, we can construct the typing relation for the substituted s and say that the type of the overall term s has been preserved after substitution. Thus, the lemma is satisfied.

Case App**:**
For this case, we let s be $b(\overrightarrow{e_i}, \overrightarrow{b_j})$. In this situation there are two sub-cases:

- If $b_1 = b$, after substitution, s would be $b[b \mapsto b_2](\overrightarrow{e_i}[b \mapsto b_2], \overrightarrow{b_j}[b \mapsto b_2])$. Therefore, we would end up with $b_2(\overrightarrow{e_i}[b \mapsto b_2], \overrightarrow{b_j}[b \mapsto b_2])$ after substitution. b_2 maintains the same type as b because $b_1 : \sigma = b : \sigma$. From the mutual induction hypothesis, we know that any expressions in $\overrightarrow{e_i}$ that contain b would be substituted while maintaining the type of $\overrightarrow{e_i}$ after substitution. Additionally, the same can be said for any blocks in $\overrightarrow{b_j}$ that match b. Thus, from the App typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

- If $b_1 \neq b$, after substitution, s would be $b[b_1 \mapsto b_2](\overrightarrow{e_i}[b_1 \mapsto b_2], \overrightarrow{b_j}[b_1 \mapsto b_2])$. Therefore, we would end up with $b(\overrightarrow{e_i}[b \mapsto b_2], \overrightarrow{b_j}[b \mapsto b_2])$ after substitution. From the mutual induction hypothesis, we know that any expressions in $\overrightarrow{e_i}$ that contain b_1 would be substituted while maintaining the type of $\overrightarrow{e_i}$ after substitution. Additionally, the same can be said for any blocks in $\overrightarrow{b_j}$ that match b_1. Thus, from the App typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

Case Def:
For this case, we let s be **def** $f = b$; s'. In this situation there are three sub-cases:

- If $b_1 = f$, after substitution, s would be **def** $f[f \mapsto b_2] = b[f \mapsto b_2]$; $s'[f \mapsto b_2]$. Therefore, we would end up with **def** $b_2 = b$; $s'[f \mapsto b_2]$ after substitution. b_2 maintains the same type as f because $b_1 : \sigma = f : \sigma$. We know that b can not be the same as f otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s' such that it preserves type after substitution. Thus, from the Def typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.
- If $b_1 = b$, after substitution, s would be **def** $f[b \mapsto b_2] = b[b \mapsto b_2]$; $s'[b \mapsto b_2]$. Therefore, we would end up with **def** $f = b_2$; $s'[b \mapsto b_2]$ after substitution. b_2 maintains the same type as b because $b_1 : \sigma = b : \sigma$. We know that b can not be the same as f otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s' such that it preserves type after substitution. Thus, from the Def typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.
- Finally, if $b_1 \neq b$ and $b_1 \neq f$, after substitution, s would be **def** $f[b_1 \mapsto b_2] = b[b_1 \mapsto b_2]$; $s'[b_1 \mapsto b_2]$. Therefore, we would end up with **def** $f = b$; $s'[b_1 \mapsto b_2]$ after substitution. We can apply the induction hypothesis to s' such that it preserves type after substitution. Thus, from the Def typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

Case SSub:
For this case, let s be s. Then, this case is trivially true because of the induction hypothesis.

Case Try:
For this case, we let s be **try** $\{f \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. In this situation there are three sub-cases:

- If $b_1 = f$, after substitution, s is **try** $\{f[f \mapsto b_2] \Rightarrow s_1[f \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[f \mapsto b_2]) \Rightarrow s_2[f \mapsto b_2]\}$. Therefore, we get **try** $\{b_2 \Rightarrow s_1[f \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2[f \mapsto b_2]\}$ after substitution. b_2 maintains the same type as f because $b_1 : \sigma = f :$

σ. We know that k can not be the same as f otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the Try typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

- If $b_1 = k$, after substitution, s is **try** $\{f[k \mapsto b_2] \Rightarrow s_1[k \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[k \mapsto b_2]) \Rightarrow s_2[k \mapsto b_2]\}$. Therefore, we get **try** $\{f \Rightarrow s_1[k \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, b_2) \Rightarrow s_2[f \mapsto b_2]\}$ after substitution. b_2 maintains the same type as k because $b_1 : \sigma = k : \sigma$. We know that k can not be the same as f otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the Try typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

- If $b_1 \neq f$ and $b_1 \neq k$, s is **try** $\{f[b_1 \mapsto b_2] \Rightarrow s_1[b_1 \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[b_1 \mapsto b_2]) \Rightarrow s_2[b_1 \mapsto b_2]\}$. Therefore, we get **try** $\{f \Rightarrow s_1[b_1 \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2[b_1 \mapsto b_2]\}$ after substitution. We can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the Try typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

Case Intercept**:**
For this case, we let s be **intercept** $\{f \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. In this situation there are three sub-cases:

- If $b_1 = f$, s is **intercept** $\{f[f \mapsto b_2] \Rightarrow s_1[f \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[f \mapsto b_2]) \Rightarrow s_2[f \mapsto b_2]\}$. Therefore, we get **intercept** $\{b_2 \Rightarrow s_1[f \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2[f \mapsto b_2]\}$ after substitution. b_2 maintains the same type as f because $b_1 : \sigma = f : \sigma$. We know that k can not be the same as f otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the Intercept typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

- If $b_1 = k$, s is **intercept** $\{f[k \mapsto b_2] \Rightarrow s_1[k \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[k \mapsto b_2]) \Rightarrow s_2[k \mapsto b_2]\}$. Therefore, we get **intercept** $\{f \Rightarrow s_1[k \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, b_2) \Rightarrow s_2[f \mapsto b_2]\}$ after substitution. b_2 maintains the same type as k because $b_1 : \sigma = k : \sigma$. We know that k can not be the same as f otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the Intercept typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

- If $b_1 \neq f$ and $b_1 \neq k$, s is **intercept** $\{f[b_1 \mapsto b_2] \Rightarrow s_1[b_1 \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[b_1 \mapsto b_2]) \Rightarrow s_2[b_1 \mapsto b_2]\}$. Thus, we get **intercept** $\{f \Rightarrow s_1[b_1 \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow$

$s_2[b_1 \mapsto b_2]\}$ after substitution. We can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the INTERCEPT typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

Case CAPINTERCEPT**:**
For this case, we let s be **intercept** $\{\mathbf{cap}_l \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$. In this situation there are three sub-cases:

- If $b_1 = \mathbf{cap}_l$, after substitution, s is **intercept** $\{\mathbf{cap}_l[\mathbf{cap}_l \mapsto b_2] \Rightarrow s_1[\mathbf{cap}_l \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[\mathbf{cap}_l \mapsto b_2]) \Rightarrow s_2[\mathbf{cap}_l \mapsto b_2]\}$. Thus, we get **intercept** $\{b_2 \Rightarrow s_1[\mathbf{cap}_l \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2[\mathbf{cap}_l \mapsto b_2]\}$ after substitution. b_2 maintains the same type as $\mathbf{cap}_l$ because $b_1 : \sigma = \mathbf{cap}_l : \sigma$. We know that k can not be the same as $\mathbf{cap}_l$ otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the CAPINTERCEPT typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.
- If $b_1 = k$, s is **intercept** $\{\mathbf{cap}_l[k \mapsto b_2] \Rightarrow s_1[k \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[k \mapsto b_2]) \Rightarrow s_2[k \mapsto b_2]\}$. Therefore, we get **intercept** $\{\mathbf{cap}_l \Rightarrow s_1[k \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, b_2) \Rightarrow s_2[\mathbf{cap}_l \mapsto b_2]\}$ after substitution. b_2 maintains the same type as k because $b_1 : \sigma = k : \sigma$. We know that k can not be the same as $\mathbf{cap}_l$ otherwise the statement would not be validly typed. Additionally, we can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the CAPINTERCEPT typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.
- Otherwise, s is **intercept** $\{\mathbf{cap}_l[b_1 \mapsto b_2] \Rightarrow s_1[b_1 \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k[b_1 \mapsto b_2]) \Rightarrow s_2[b_1 \mapsto b_2]\}$. Thus, after substitution, we get **intercept** $\{\mathbf{cap}_l \Rightarrow s_1[b_1 \mapsto b_2]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2[b_1 \mapsto b_2]\}$. We can apply the induction hypothesis to s_1 and s_2 to show that these two statements maintain their type after substitution. Thus, from the CAPINTERCEPT typing rule, we can construct a typing relation that tells us that s maintains type after substitution. Thus, the lemma is satisfied.

PROOF. The following are for the mutual induction cases on blocks where we prove that if $\Gamma, b_1 : \sigma' \mid \Sigma \vdash b : \sigma \mid C$ and $\Gamma \mid \Sigma \vdash b_2 : \sigma \mid C$, then $\Gamma \mid \Sigma \vdash b[b_1 \mapsto b_2] : \sigma \mid C$.

Case TRANSPARENT **and** TRACKED**:**
For these cases, we let b be f. In this situation, there are two sub-cases:

- If $b_1 = f$, b would be $f[f \mapsto b_2]$. Thus, after substitution, b would be b_2. We know that b_2 maintains the type of f because $b_1 : \sigma = f : \sigma$. Thus, from the TRANSPARENT rule, we can construct the typing relation for the substituted b and say that the type of the overall term b has been preserved after substitution. Thus, the lemma is satisfied.

- If $b_1 \neq f$, b would be $f[b_1 \mapsto b_2]$. Thus, after substitution, b would be f. Because there is no substitution, the desired result is immediate. Thus, the lemma is satisfied.

Case Block**:**
For this case, we let b be $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j : \sigma_j}) \Rightarrow s'\}$. Therefore, after substitution, b would be $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j[b_1 \mapsto b_2] : \sigma_j}) \Rightarrow s'[b_1 \mapsto b_2]\}$. From the induction hypothesis, we know that blocks $(\overrightarrow{g_j})$ and s' maintain their types after substitution. Thus, from the Block rule, we can say that the type of the overall term b has been preserved after substitution. Thus, the lemma is satisfied.

Case BoxElim**:**
For this case, we let b be **unbox** e. After substitution, b would be **unbox** $e[b_1 \mapsto b_2]$. From the mutual induction hypothesis, we know that e maintains it type after substitution. Thus, from the BoxElim rule, we can construct the typing relation for the substituted b and say that the type of the overall term b has been preserved after substitution. Thus, the lemma is satisfied.

Case BSub**:**
For this case, we let b be b. This is trivially true because of the induction hypothesis.

Proof. The following are for the mutual induction cases on blocks where we prove that if $\Gamma, b_1 : \sigma \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash b_2 : \sigma \mid C$, then $\Gamma \mid \Sigma \vdash e[b_1 \mapsto b_2] : \tau$.

Case Lit**:**
For this case, we let e be n. This case is trivial because b_1 cannot take the form of n where n is an integer. Thus, there can be no substitution and the result is immediate. Thus, the lemma is satisfied.

Case Var**:**
For this case, we let e be x where $x : \tau \in \Gamma$. This case is trivial because b_1 cannot take the form of x because x is a primitive value. Thus, there can be no substitution and the result is immediate. Thus, the lemma is satisfied.

Case BoxIntro**:**
For this case, we let e be **box** b. There are two sub-cases in this situation:

- If $b_1 = b$, e would be **box** $b[b \mapsto b_2]$. Thus, after substitution, e would be **box** b_2. We know that b_2 maintains the type of b because $b_1 : \sigma = b : \sigma$. Thus, from the BoxIntro rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.

- If $b_1 \neq b$, after substitution, e would be **box** $b[b_1 \mapsto b_2]$. From the mutual induction hypothesis, we know that b maintains it type after substitution. Thus, from the BoxIntro rule, we can construct the typing relation for the substituted e and say that the type of the overall term e has been preserved after substitution. Thus, the lemma is satisfied.

Case DEREF**:**
For this case, we let e be $!e$. This case is trivial because b_1 cannot take the form of e. Thus, there can be no substitution and the result is immediate. Thus, the lemma is satisfied.

Case NEW**:**
For this case, we let e be **new** e. This case is trivial because b_1 cannot take the form of e. Thus, there can be no substitution and the result is immediate. Thus, the lemma is satisfied.

Case ASSIGN**:**
For this case, we let e be $e_1 \coloneqq e_2$. This case is trivial because b_1 cannot take the form of e_1 or e_2. Thus, there can be no substitution and the result is immediate. Thus, the lemma is satisfied.

Case REF**:**
For this case, we let e be l. This case is trivial because b_1 cannot take the form of l. Thus, there can be no substitution and the result is immediate. Thus, the lemma is satisfied.

Lemma 4 (SUBSTITUTION OF BLOCKS AND CAPABILITY SETS)**.** *Given a well-typed statement* $\Gamma_1, f :^* \sigma, \Gamma_2 \mid \Sigma \vdash s : \tau \mid C_1$ *and a block* $E \mid \Sigma \vdash b : \sigma \mid C_2$ *that can be checked under restriction* $E \mid \Sigma \vdash C_2$ *wf, then* $\Gamma_1, \Gamma_2[f \mapsto C_2] \mid \Sigma \vdash s[f \mapsto b, f \mapsto C_2] : \tau[f \mapsto C_2] \mid C_1[f \mapsto C_2]$.

PROOF. We lift this lemma and proof directly from Brachthäuser et al. (2022)'s paper so we will reiterate the proof in too much detail. Instead, we will provide a brief outline of the proof. The main idea of this proof is that mutual induction is performed over the typing derivation much like a regular substitution lemma. The only difference in this lemma for simultaneous substitution is that simultaneous substitution requires reasoning about subset inclusion.

7.2.2 Preservation Proof

After establishing these lemmas, we can now begin the actual proof of the property of preservation.

Theorem 2 (PRESERVATION OF STATEMENTS)**.** *If* $\langle \varnothing \mid \Sigma \vdash s : \tau \mid \varnothing \mid \Xi \rangle$ *and* $s \longmapsto s'$ *then* $\langle \varnothing \mid \Sigma' \vdash s' : \tau \mid \varnothing \mid \Xi' \rangle$*. By induction on a derivation of* $s : \tau$*, we assume that the desire property holds for all sub-derivations. We approach this proof by using case analysis on the final rule in the derivation.*

Case VAL**:** $\langle$**val** $x = s_0;\ s_1 : \tau_1 \mid \Xi \rangle$ where $s_0 : \tau_0$, $s_1 : \tau_1$
For a generic s, if s has the form **val** $x = s_0;\ s_1$ for some x, s_0 and s_1. We now have two

sub-cases to consider involving s_0. This is because both x and s_1 have no bearing on the evaluation of s no matter what form they take as long as they are valid.

- If s_0 takes the form of **return** v, we can step to $s_1[x \mapsto v]$. As such the resulting term s_1 would have type τ_1 which is exactly the type of the original program s. Additionally, from the substitution lemma (Lemma 2), we know that s preserves type after substitution. Thus, the property of preservation holds.
- If s_0 takes any other valid form, according to the evaluation context $\textbf{val}\ x = E;\ s$, we would perform a step such that $E[s'] \mapsto E[s'']$ where s'' is the result of the inner evaluation step. As such, the overall type of the program s will remain the same.

Case Ret**:** $\langle \textbf{return}\ v : \tau \mid \Xi \rangle$
Trivially true because **return** v is not able to step to anything.

Case App**:** $\langle b(\overrightarrow{e_i}, \overrightarrow{b_j}) : \tau \mid \Xi \rangle$ where $b : (\overrightarrow{\tau_i}, \overrightarrow{f_j : \sigma_j}) \to \tau$
If a generic term s is of the form $b(\overrightarrow{e_i}, \overrightarrow{b_j})$, we will only need to consider the case where b is of the form $\{(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s'\}$. This is because this is the only form b could take and still be validly typed. As such, we only need to consider the evaluation rule *(app)* where s steps to $s'[\overrightarrow{x_i \mapsto v_i}, \overrightarrow{f_j \mapsto C_j}, \overrightarrow{f_j \mapsto w_j}]$. We now apply the lemma for simultaneous substitution (Lemma 4) which says that the type of the statement s remains the same even after simultaneous substitution of blocks and capability sets is applied. From the App and Block typing rules, we can determine that s' has the type τ. Thus, this now satisfies the property of preservation. Therefore, we can state that:

$$\langle b(\overrightarrow{e_i}, \overrightarrow{b_j}) : \tau \mid \Xi \rangle \longmapsto \langle s'[\overrightarrow{x_i \mapsto v_i}, \overrightarrow{f_j \mapsto C_j}, \overrightarrow{f_j \mapsto w_j}] : \tau \mid \Xi' \rangle$$

Case Def**:** $\langle \textbf{def}\ f = b;\ s_1 : \tau \mid \Xi \rangle$ where $b : \sigma,\ s_1 : \tau$
For a generic term s, if s is of the form $\textbf{def}\ f = b;\ s_1$, we must consider a few sub-cases for b.

- If b is of the form f or of the form **unbox** e, s would not be well-typed. Thus, we can safely ignore these sub-cases.
- If b is of the form $\left\{\left(\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j} \Rightarrow s_2\right)\right\}$, we can apply the **def** evaluation rule. This would result in a step to $s_2[f \mapsto w]$. From the substitution lemma for blocks (Lemma 3), s_2 will preserve its type after substitution. As a result, the type of s_2 would be τ. Thus, the type is maintained as desired.

Case Try**:** $\langle \textbf{try}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\} : \tau \mid \Xi \rangle$ where $s_1 : \tau,\ s_2 : \tau$
When a generic term s is of the form $\textbf{try}\ \{f \Rightarrow s_1\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$, we can apply the *(try)* evaluation rule regardless of the form of s_1 or s_2 as long as the overall term s is valid. We thus end up with $\#_l\{s_1[f \mapsto \{l\}, f \mapsto \textbf{cap}_l]\}\ \textbf{with}\ \{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$ after a step is taken. From the lemma of simultaneous substitution (Lemma 4), we know

that the type of s_1 will remain the same after substitution. Therefore, from the Reset typing rule we know that $\#_l\{s_1[f \mapsto \{l\}, f \mapsto \textbf{cap}_l]\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$ has a type of τ. Thus, this satisfies the preservation of type.

Case SSub**:** $\langle s : \tau \mid \Xi \rangle$
When s is of the form s, the case is trivially true because of the induction hypothesis.

Case Cap**:** $\langle \textbf{cap}_l : \tau_i \to \tau_0 \mid \Xi \rangle$
Trivial because a generic term s of the form $\textbf{cap}_l$ cannot step to anything. Thus, this case is vacuously true.

Case Reset**:** $\langle \#_l\{s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\} : \tau \mid \Xi \rangle$ where $s_1 : \tau$, $s_2 : \tau$
When s is of the form $\#_l\{s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$, the only evaluation rule which applies is *(cap)*. When this rule is applied, the evaluated program would be:

$$s_2[\overrightarrow{x_i \mapsto v_i},\ k \mapsto \{\ y \Rightarrow \#_l\{\ \mathbf{H}_l[\textbf{return}\ y]\ \}\ \textbf{with}\ h\ \}]$$

To prove preservation of types, we first have to construct the typing derivation of this evaluated program. Here, we draw on the outline provided by Brachthäuser et al. (2022). By assuming that $l : \overrightarrow{\tau_i} \to \tau_0 \in \Xi$, we can apply the substitution lemma on k. Thus, we just need to show that $\varnothing \vdash \{\ y \Rightarrow \#_l\{\ \mathbf{H}_l[\textbf{return}\ y]\ \}\ \textbf{with}\ h\ \} : \tau_0 \to \tau \mid C$. This can be done by applying the typing rules of Block and Reset and also using Lemma 5 which is directly quoted from Brachthäuser et al. (2022).

Lemma 5. *Given a well-typed effect call* $\varnothing \vdash H_l[\textbf{cap}_l(\overrightarrow{v_i})] : \tau \mid C \cup \{l\}$ *with effect signature* $l : \overrightarrow{\tau_i} \to \tau_0 \in \Xi$*, it follows that* $y : \tau_0 \vdash \mathbf{H}_l[\textbf{return}\ y] : \tau \mid C \cup \{l\}$.

Case Intercept**:** $\langle$**intercept** $\{f \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\} : \tau \mid \Xi \rangle$ where $s_1 : \tau$, $s_2 : \tau$
We do not need to consider this case because it does not type check under an empty Γ. Even as we build our proof inductively, there will be no case where we need to consider a non-empty Γ.

Case CapIntercept**:** $\langle$**intercept** $\{\textbf{cap}_l \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\} : \tau \mid \Xi \rangle$ where $s_1 : \tau$, $s_2 : \tau$
When a generic term s is of the form **intercept** $\{\textbf{cap}_l \Rightarrow s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$, the only evaluation rule which applies is *(interc)*. The resulting, evaluated program $\#_l\{s_1\}$ **with** $\{(\overrightarrow{x_i}, k) \Rightarrow s_2\}$ has a type of τ according to the Reset typing rule. As such, the property of preservation holds.

Theorem 3 (Preservation of Blocks)**.** *If* $\langle \varnothing \mid \Sigma \vdash b : \sigma \mid C \mid \Xi \rangle$ *and* $\langle b \mid \Xi \rangle \longmapsto \langle b' \mid \Xi' \rangle$ *then* $\langle \varnothing \mid \Sigma' \vdash b' : \sigma \mid C \mid \Xi' \rangle$ *for an arbitrary, well-typed* e, τ, Ξ *and* Σ. *By induction on a derivation of* $s : \tau$*, we assume that the desire property holds for all sub-derivations. We approach this proof by using case analysis on the final rule in the derivation.*

Case BoxElim**:** $\langle$**unbox** $e : \sigma \mid \Xi \rangle$

When a generic term b is of the form **unbox** e, e must be of the form **unbox** (**box** b') otherwise b would be ill-typed. Thus, the only reduction rule which applies here is *(box)*. Under this reduction rule, we would step to b'. According to the BoxIntro typing rule, we know that b' is of type σ. Thus, the property of preservation holds.

Case BSub**:** $\langle b : \sigma \mid \Xi \rangle$
Trivially true because of the induction hypothesis.

Case Block**:** $\langle \{(\overrightarrow{x_i : \tau_i}, \overrightarrow{g_j : \sigma_j}) \Rightarrow s\} : (\overrightarrow{\tau_i}, \overrightarrow{g_j : \sigma_j}) \rightarrow \tau \mid \Xi \rangle$
Trivially true because this case is not able to step anywhere.

Case Transparent **and** Tracked**:** $\langle f : \sigma \mid \Xi \rangle$ and $\langle f : \sigma \mid \Xi \rangle$
We do not need to consider these cases because they do not type check under an empty Γ. As such, even as we build our proof inductively, there will be no case where we need to consider a non-empty Γ.

Theorem 4 (Preservation of Expressions)**.** *If* $\langle \varnothing \mid \Sigma \vdash e : \tau \mid \Xi \rangle$ *and* $\langle e \mid \Xi \rangle \longmapsto \langle e' \mid \Xi' \rangle$ *then* $\langle \varnothing \mid \Sigma' \vdash e' : \tau \mid \Xi' \rangle$ *for an arbitrary, well-typed* e, τ, Ξ *and* Σ*. By induction on a derivation of* $e : \tau$*, we assume that the desire property holds for all sub-derivations. We approach this proof by using case analysis on the final rule in the derivation.*

Case Deref**:** $\langle !e_1 : \tau \mid \Xi \rangle$
When a generic term e is of the form $!e_1$, the only reduction rule which could apply is *(deref)*. This would also constrain e_1 to be of the form l and specify that $\Xi(l) = v$. Thus, this l would have the type **Ref** τ where $\Sigma(l) = \tau$. Because Ξ is well-typed, by definition of well-typed stores (Pierce, 2002), we can say that $\Gamma \mid \Sigma \ \vdash \Xi(l) : \Sigma(l), \forall l \in \mathit{dom}(\Xi)$. In other words, it is true that $\Gamma \mid \Sigma \vdash v : \tau$. Thus, even after the *(deref)* reduction rule is applied and e steps to v, the type of e is preserved. Thus the induction hypothesis holds.

Case New**:** $\langle$**new** $e_1 :$ **Ref** $\tau \mid \ \Xi \rangle$
When a generic term e is of the form **new** e_1, the only reduction rule which applies is *(new)*. This means that e_1 is of the form v and e steps to l. According to the Ref typing rule, the type of l is **Ref** τ where $\Sigma(l) = \tau$. From the definition of a well-typed store Ξ (Pierce, 2002), we know that $\Gamma \mid \Sigma \ \vdash \Xi(l) : \Sigma(l), \forall l \in \mathit{dom}(\Xi)$. As such, we know that because $\Xi(l) = v$, we can say that $v : \tau$. Through the New typing rule, we know that the overall type of a **new** e is **Ref** τ where $e : \tau$. Thus, given that $v : \tau$, we can say that **new** v has type **Ref** τ where τ is the same τ in $l :$ **Ref** τ. Therefore, after stepping via the *(new)* rule, the type still remains the same. Thus, the induction hypothesis holds.

Case Assign**:** $\langle e_1 \coloneqq e_2 : \tau \mid \Xi \rangle$
When a generic term e is of the form $e_1 \coloneqq e_2$, the only reduction rule which is able to be applied is *(assign)*. This would mean that e_1 is of the form l and e_2 is of the form v. Additionally, this reduction rule specifies that $\langle e_1 \coloneqq e_2 \mid \Xi \rangle \longmapsto \langle v \mid \Xi[l \mapsto v] \rangle$ From the Assign typing rule, we know that e_2 has type τ. Thus, after stepping to $v : \tau$, e maintains the same type of τ. Therefore, the induction hypothesis holds.

Case $\textsc{Var}$**:** $\langle x : \tau \mid \Xi \rangle$
We do not need to consider this case because it does not type check under an empty Γ. As such, even as we build our proof inductively, there will be no case where we need to consider a non-empty Γ.

Case $\textsc{Lit}$**,** $\textsc{BoxIntro}$**, and** $\textsc{Ref}$**:** $\langle n : \mathrm{Int} \mid \Xi \rangle$, $\langle \textbf{box}\ b : \sigma\ \textbf{at}\ C \mid \Xi \rangle$, and $\langle l : \textbf{Ref}\ \tau \mid \Xi \rangle$
Trivially true because the generic term e under any of these cases can not step to anything.

Chapter 8

Discussion

Due to the fact that this thesis introduces an approach to programming languages research as well as running case studies, Chapter 8 will be delineated into two separate sections. The first section will discuss the overall methodology of the thesis in addition to the novel approach which is introduced. Following this, we will examine the case studies under the lens of the actual additions which were made to the System C language. Specifically, we will evaluate the case studies by their potential applications. Let us first begin with the discussion on the methodology and approach.

8.1 Methodology and Approach

8.1.1 Methodology

As a whole, the methodology presented was only a vehicle to be able to accurately and thoroughly investigate the Redex augmented approach which has been introduced in this thesis. Thus, we will only briefly examine the effectiveness of the methodology in enabling us to fully explore the novel approach's effect on language research.

Overall, the methodology was designed to facilitate a smooth and efficient way to study Redex modeling's impact on language research. In this capacity, the desired outcome was achieved. By starting with a base model of System C in Redex, we were able to form a foundation for case studies to be performed. Furthermore, these case studies allowed for the approach to be examined in a more objective manner. The case studied performed form the basis of further discussion which is had in Chapter 8.1.2.

Obviously, there are also limitations with this methodology. For one, it lacks empirical evidence for the effectiveness of adding Redex in to a language research approach. If possible, future work that builds upon this thesis should attempt to perform surveys on participating researchers to gather real data on if this approach helps with research.

On top of this, the case studies presented in this thesis could have been more robust in terms of quantity and complexity. More complex case studies would have been able to better explore the nuances of modeling languages before proving them. However, as it is, the methodology is still able to adequately express the results of using our novel approach.

8.1.2 Approach

The approach put forward in this thesis contains two general parts. First, the language which is to be researched and proved is implemented in Redex. After this implementation, a formal proof of safety for the language is done. Our discussion of our novel approach is mainly informed by the two case studies which are carried out in Chapter 5 and Chapter 6.

From the case studies which we conducted, we can draw a comparison between our approach with a traditional approach to language research that does not use modeling. As opposed to the regular workflow where a researcher transitions straight from designing new features into a proof, having a Redex prototype allows for a more iterative design process. In particular, the ability for examples to be run and type checked in real-time via Redex benefits the language design process because less time is spent confirming the behaviour of new language features. Researchers are able to offload much of the time expensive process of running through example cases by just allowing Redex to model the outcomes. This can be seen in both case studies where we were able to run examples mechanically via the Redex model instead of having to resort to dry run testing. As as result of the efficiency of running and type checking examples in Redex, intuition for language features can be developed much faster than if examples were checked by hand.

Once intuition has been gained from a working Redex implementation, the next step of the approach involves a proof of safety. While this step coincides with the conventional work flow in language research, the safety proof also benefits from the existence of a Redex model. A Redex implementation of an unproven language is able to preemptively catch errors in language design which may result in unsafe behaviour. Such a preventative safeguard ensures that less time is wasted on trivial errors in language design. This is exhibited in the second case study in Chapter 6 where Redex is used to catch a design flaw. Additionally, there are instances in the first case study in Chapter 5 where the Redex modeling was able to catch errors in the design of the reduction rules. Furthermore, if the proof of safety does expose a flaw with the language, a Redex model would be able to help with identifying where exactly this flaw may lie. As long as the model accurately reflects language behaviour, the researcher only has to run sufficiently many example cases to test which part of the language may be unsafe.

Another advantage which this approach seems to offer over a regular workflow without model assistance is the ability for a more agile research process. Typically, when attempting programming languages research, a researcher will design language features and then attempt to prove that these features are safe. However, when the language feature is complex or a combination of smaller language features, progress on proving the safety of the language feature can be time expensive. A decision will often have to be made between proving safety for a language feature in its entirety and designing and proving pieces of the language feature in a piecemeal manner. The first method could result in an error being discovered at the very end while the second method takes a long amount of time. A Redex model simplifies this dilemma by allowing some of the brunt of the iteration process to be taken by the model itself. This streamlined methodology is displayed in both our case studies where we are able to leave the proof of safety until after both the case studies were completed instead of having to iteratively attempt a safety proof. Thus, our proposed approach allows for the shorter time frame of only performing a proof of safety at the very end of a complex language feature design while also ensuring that the language feature which is designed is less likely to be unsafe.

Any discussion of this approach would be remiss to not contain some of the limitations which the approach suffers from. To start, it must be noted that implementing a language or language features within Redex does require time and some understanding of how Redex functions. As such, this is a trade off which must be considered when using this approach. However, for complex languages, most of the time spent initially implementing the language in Redex is offset by the time saved through the automation of program evaluation and type checking. Furthermore, there may be some mechanisms which are easier to design than to implement in a Redex model. This may result in some language features being easier to prove (via some convenient hand-waving) than to model in Redex. Thus, this will need to be a trade-off which is considered before adopting our approach.

Overall, the approach proposed in this thesis allows for faster iteration than doing a proof without modeling while having the benefits of error detection and gained intuition. It must be said that this approach has trade offs such as in terms of time required for a Redex implementation. However, modeling can bring benefits to programming languages research if language features need to be proved.

8.2 Language Extensions

To evaluate the actual language features which were added to System C under our case studies, we consider the potential applications and benefits of the features added.

Effect handler abstraction. The addition of an interception mechanism to the effects handler presented in the original System C language introduces a layer of abstraction to the handler. There are many potential applications to this abstraction. For one, programs which are concerned with security could benefit from the ability to intercept an effect before the associated effect handler executes code. In situations where the effect handler code may not necessarily be vetted, the interception could allow for a way to avoid running foreign code and thus prevent code injection attacks. An example of when this could occur would be when foreign libraries are used. This benefit of abstraction could be extended to any number of security applications where sensitive data is at stake.

Language usability. A prime benefit which rises from the addition of variables in the System C language is language ergonomics and usability. Almost all popular languages have a grammatical structure which represents a variable. This is because variables are a very convenient construct which allow programs to share state easily (Van Der Werf et al., 2023). As such, with the addition of variables to System C, we unlock the ability to easily store and manipulate data - a core tenant of many non-functional programs.

Chapter 9

Conclusion

In this thesis, we present a modified approach to conducting research into programming languages and, specifically, language proofs. This method involves using Redex, a library in the Racket programming language, to model language features or even entire languages in order to allow for more intuitive and less tedious proving of language safety. To demonstrate the effectiveness of our new approach, we conduct several case studies on the System C language by extending the language with certain features including variables and handler abstraction. These case studies are motivated by a variety of reasons ranging from wanting to extend the usability of algebraic handlers to completing ideas which were first established in Ou (2023). We then discuss to viability of our approach and compare our Redex based method against the conventional methodologies that are employed when doing research. In the end, we conclude that our approach has certain benefits over the traditional ways of proving safety but further research may still be required. We now present some future lines of efforts which can potentially extend the ideas presented in this paper.

9.1 Future Work

The ideas explored in this paper can be extended in a variety of ways which may be interesting to the field of programming languages. We list some of these potential extensions below.

Adding Coq to our approach. Although the basis of the approach which we presented in this thesis was motivated in part by trying to avoid the rigidity of Coq, mechanised proofs are still a powerful tool in programming languages research. As such, future iterations of our approach may try to incorporate mechanised provers such as Coq in some capacity. This could manifest simply through a mechanised proof as an additional step at the end of the approach or perhaps something more involved.

Surveying the usefulness of our approach. The approach presented in this thesis is accompanied by two case studies which demonstrate its effectiveness. However, this is only one such example of the approach in use. It may be pertinent to conduct surveys or experiments to collect data on how useful the approach actually is for a diverse range of relevant stakeholders. These stakeholders could have varying background in programming languages research from undergraduate students to published and experienced researchers. The outcomes of this survey may also be able to inform further improvements to our approach.

Further investigating the combination of abstract and algebraic effects. Through our case studies, we extended the System C language. One of the additions made to the language involved the inclusion of an interception mechanism to the existing algebraic effect handler in System C. This extension to the language was inspired by Ou (2023)'s paper which attempted to combine abstract and algebraic effects. Although we managed to introduce some light abstraction in this paper, most of the abstraction is orthogonal to the algebraic effects handler because this was not the main focus of our thesis. As such, a future avenue of research may be to try and extend System C in a way which more comprehensively abstracts the algebraic effects handler.

Bibliography

Abelson, H. and Sussman, G. J., 1996. *Structure and interpretation of computer programs*. The MIT Press. http://library.oapen.org/handle/20.500.12657/26092. [Cited on page 10.]

Barga, R.; Lomet, D.; Shegalov, G.; and Weikum, G., 2004. Recovery guarantees for internet applications. *ACM Trans. Internet Technol.*, 4, 3 (Aug. 2004), 289–328. doi:10.1145/1013202.1013205. https://doi.org/10.1145/1013202.1013205. [Cited on page 7.]

Blanco, R.; Miller, D.; and Momigliano, A., 2019. Property-based testing via proof reconstruction. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19 (Porto, Portugal, 2019). Association for Computing Machinery, New York, NY, USA. doi:10.1145/3354166.3354170. https://doi-org.virtual.anu.edu.au/10.1145/3354166.3354170. [Cited on page 6.]

Brachthäuser, J. I.; Schuster, P.; Lee, E.; and Boruch-Gruszecki, A., 2022. Effects, capabilities, and boxes: From scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.*, 6, OOPSLA1 (apr 2022). doi:10.1145/3527320. https://doi.org/10.1145/3527320. [Cited on pages 7, 21, 24, 51, 53, 65, and 67.]

Brachthäuser, J. I.; Schuster, P.; and Ostermann, K., 2020. Effekt: Lightweight effect polymorphism for handlers. Technical report, Technical Report. University of Tübingen, Germany. [Cited on page 6.]

Craig, A.; Potanin, A.; Groves, L.; and Aldrich, J., 2018. Capabilities: Effects for free. In *Formal Methods and Software Engineering*, 231–247. Springer International Publishing, Cham. [Cited on page 7.]

Cremers, A. and Ginsburg, S., 1975. Context-free grammar forms. *Journal of Computer and System Sciences*, 11, 1 (1975), 86–117. doi:https://doi.org/10.1016/S0022-0000(75)80051-1. https://www.sciencedirect.com/science/article/pii/S0022000075800511. [Cited on page 21.]

Dennis, J. B. and Van Horn, E. C., 1966. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9, 3 (Mar. 1966), 143–155. doi:10.1145/365230.365252. https://doi.org/10.1145/365230.365252. [Cited on page 7.]

Emre, M.; Schroeder, R.; Dewey, K.; and Hardekopf, B., 2021. Translating c to safer rust. *Proc. ACM Program. Lang.*, 5, OOPSLA (Oct. 2021). doi:10.1145/3485498. https://doi.org/10.1145/3485498. [Cited on page 1.]

Fahndrich, M. and DeLine, R., 2002. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02 (Berlin, Germany, 2002), 13–24. Association for Computing Machinery, New York, NY, USA. doi:10.1145/512529.512532. https://doi.org/10.1145/512529.512532. [Cited on page 1.]

Findler, R. B.; Klein, C.; Fetscher, B.; and Felleisen, M., 2015. Redex: Practical semantics engineering. *Online at http://docs. racket-lang. org/redex*, (2015). [Cited on page 5.]

Flatt, M. and PLT, 2010. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc. https://racket-lang.org/tr1/. [Cited on page 1.]

Gazi, O., 2024. Modern c programming. (2024). doi:10.1007/978-3-031-45361-8. https://doi.org/10.1007/978-3-031-45361-8. [Cited on page 9.]

Gosling, J.; Joy, B.; Steele, G.; and Bracha, G., 2005. *The Java Language Specification, Third Edition*, 688. [Cited on page 6.]

Jim, T.; Morrisett, J.; Grossman, D.; Hicks, M.; Cheney, J.; and Wang, Y., 2002. Cyclone: A safe dialect of c. 275–288. [Cited on page 1.]

Kiselyov, O. and Sivaramakrishnan, K., 2018. Eff directly in ocaml. *Electronic Proceedings in Theoretical Computer Science*, 285 (Dec. 2018), 23–58. doi:10.4204/eptcs.285.2. http://dx.doi.org/10.4204/EPTCS.285.2. [Cited on page 6.]

Klabnik, S. and Nichols, C., 2023. *The Rust programming language*. No Starch Press. [Cited on page 1.]

Klein, C., 2009. *Experience with randomized testing in programming language metatheory*. Ph.D. thesis, Master's thesis, Northwestern, August 2009. http://plt. eecs. northwestern .... [Cited on page 6.]

Klein, C.; Clements, J.; Dimoulas, C.; Eastlund, C.; Felleisen, M.; Flatt, M.; McCarthy, J. A.; Rafkind, J.; Tobin-Hochstadt, S.; and Findler, R. B., 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '12 (Philadelphia, PA, USA, 2012), 285–296. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2103656.2103691. https://doi.org/10.1145/2103656.2103691. [Cited on page 6.]

Klein, C. and Findler, R. B., 2009. Randomized testing in plt redex. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*. [Cited on page 5.]

Kurilova, D.; Potanin, A.; and Aldrich, J., 2014. Wyvern: Impacting software security via programming language design. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14 (Portland, Oregon, USA, 2014), 57–58. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2688204.2688216. https://doi-org.virtual.anu.edu.au/10.1145/2688204.2688216. [Cited on page 6.]

Li, P., 2004. Safe systems programming languages. https://api.semanticscholar.org/CorpusID:2448687. [Cited on page 1.]

Lindley, S.; Matache, C.; Moss, S.; Staton, S.; Wu, N.; and Yang, Z., 2024. Scoped effects as parameterized algebraic theories. In *Programming Languages and Systems*, 3–21. Springer Nature Switzerland, Cham. [Cited on page 7.]

Lucassen, J. M. and Gifford, D. K., 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88 (San Diego, California, USA, 1988), 47–57. Association for Computing Machinery, New York, NY, USA. doi:10.1145/73560.73564. https://doi.org/10.1145/73560.73564. [Cited on pages 6 and 7.]

Macabeus, B.; Albuquerque, M.; and Pace, E., 2020. Project title. https://github.com/macabeus/js-proposal-algebraic-effects. [Cited on page 7.]

Mackay, J.; Potanin, A.; Aldrich, J.; and Groves, L., 2019. Decidable Subtyping for Path Dependent Types. *Proc. ACM Program. Lang.*, 4, POPL (Dec. 2019). doi:10.1145/3371134. https://doi.org/10.1145/3371134. Place: New York, NY, USA Publisher: Association for Computing Machinery. [Cited on page 6.]

McCracken, D. D. and Reilly, E. D., 2003. *Backus-Naur form (BNF)*, 129–131. John Wiley and Sons Ltd., GBR. ISBN 0470864125. [Cited on page 22.]

Melicher, D.; Xu, A.; Zhao, V.; Potanin, A.; and Aldrich, J., 2022. Bounded abstract effects. *ACM Trans. Program. Lang. Syst.*, 44, 1 (jan 2022). doi:10.1145/3492427. https://doi.org/10.1145/3492427. [Cited on page 1.]

Milner, R., 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 3 (1978), 348–375. doi:https://doi.org/10.1016/0022-0000(78)90014-4. https://www.sciencedirect.com/science/article/pii/0022000078900144. [Cited on page 1.]

Nawaz, M. S.; Malik, M.; Li, Y.; Sun, M.; and Lali, M. I. U., 2019. A survey on theorem provers in formal methods. https://arxiv.org/abs/1912.03028. [Cited on page 14.]

Nistor, L.; Kurilova, D.; Balzer, S.; Chung, B.; Potanin, A.; and Aldrich, J., 2013. Wyvern: a simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and InHerItance*, MASPEGHI '13 (Montpellier, France, 2013), 9–16. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2489828.2489830. https://doi.org/10.1145/2489828.2489830. [Cited on page 6.]

Odersky, M.; Altherr, P.; Cremet, V.; Emir, B.; Maneth, S.; Micheloud, S.; Mihaylov, N.; Schinz, M.; Stenman, E.; and Zenger, M., 2004. An overview of the scala programming language. https://infoscience.epfl.ch/handle/20.500.14299/214698. [Cited on page 1.]

Orchard, D. and Petricek, T., 2014. Embedding effect systems in haskell. *SIGPLAN Not.*, 49, 12 (Sep. 2014), 13–24. doi:10.1145/2775050.2633368. https://doi.org/10.1145/2775050.2633368. [Cited on page 6.]

Ou, G., 2023. *Relationship between Algebraic and Abstract effect.* Bachelor's thesis, Australian National University, Canberra ACT 2601. [Cited on pages 2, 19, 75, and 76.]

Ouyang, Z., 2023. *Algebraic Effects in Wyvern.* Comp3770 paper, Australian National University, Canberra ACT 2601. [Cited on pages 2 and 19.]

Patterson, D.; Mushtak, N.; Wagner, A.; and Ahmed, A., 2022. Semantic soundness for language interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022 (San Diego, CA, USA, 2022), 609–624. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3519939.3523703. https://doi.org/10.1145/3519939.3523703. [Cited on page 13.]

Pierce, B. C., 2002. *Types and Programming Languages.* The MIT Press, 1st edn. ISBN 0262162091. [Cited on pages 14, 33, 35, and 68.]

Plotkin, G. and Pretnar, M., 2009. Handlers of algebraic effects. In *Programming Languages and Systems*, 80–94. Springer Berlin Heidelberg, Berlin, Heidelberg. [Cited on page 7.]

Potanin, A.; Noble, J.; Clarke, D.; and Biddle, R., 2004. Defaulting Generic Java to Ownership. In *FTfJP*. Springer-Verlag, Oslo, Norway. [Cited on page 6.]

Syme, D., 1999. *Proving Java Type Soundness*, 83–118. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-48737-1. doi:10.1007/3-540-48737-9_3. https://doi.org/10.1007/3-540-48737-9_3. [Cited on page 1.]

Tanter, E., 2009. Beyond static and dynamic scope. In *Proceedings of the 5th Symposium on Dynamic Languages*, DLS '09 (Orlando, Florida, USA, 2009), 3–14. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1640134.1640137. https://doi.org/10.1145/1640134.1640137. [Cited on page 9.]

Team, T. C. D., 2024. The coq proof assistant. doi:10.5281/zenodo.11551307. https://doi.org/10.5281/zenodo.11551307. [Cited on page 14.]

The University of Texas at Austin, 2008. Exception handling. https://www.cs.utexas.edu/~mitra/csSpring2008/cs313/lectures/excep.html. [Cited on page 10.]

The Wyvern Developers, 2018. The wyvern programming language. https://wyvernlang.github.io/. Accessed: 2023-10-15. [Cited on page 2.]

Turbak, F. and Gifford, D., 2008. *Design concepts in programming languages*. MIT press. [Cited on page 6.]

Van Der Werf, V.; Zhang, M. Y.; Aivaloglou, E.; Hermans, F.; and Specht, M., 2023. Variables in practice. an observation of teaching variables in introductory programming moocs. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023 (Turku, Finland, 2023), 208–214. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3587102.3588857. https://doi.org/10.1145/3587102.3588857. [Cited on page 74.]

Wright, A. and Felleisen, M., 1994. A syntactic approach to type soundness. *Information and Computation*, 115, 1 (1994), 38–94. doi:https://doi.org/10.1006/inco.1994.1093. https://www.sciencedirect.com/science/article/pii/S0890540184710935. [Cited on page 13.]