

**Studies Concerning the Meaning of
Computer Programs:
Formal Specifications and Implementations,
Monolithic and Distributed Programs,
and the Semantics of Syntactic Transformations**

Xueying Qin

秦雪莹



Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2024

Abstract

This doctoral thesis presents three studies that concerns the meaning of computer programs in three different aspects, namely, specifications and implementations, monolithic and distributed frameworks, and the formal semantics of syntactic transformations. In general, three conceptual questions have been asked and addressed by these three studies.

The first conceptual question is: How to design a better abstraction mechanism that allows programmers to effectively express *what* they want a computer to do via some declarative yet accurate *specifications* instead of how a computer should accomplish a task via some concrete *implementations*? The study, “*Primrose: Selecting Container Data Types by Their Properties*”, addresses this question by exploring ways to design a specification for a container that truly allows a programmer to separate its interface and usage from the implementation and to select a concrete implementation determined by its interface and usage. By conducting this study, I conclude that such separation of concerns enables automation and optimisation for application developers to use a container data type in their programs.

The second conceptual question is: How to intuitively understand *distributed programs* using the same conceptual model as *monolithic programs*? The question is addressed by the study of designing a remote procedural call library — the Universal Method Invocation (UMI) library — for Rust. The UMI library supports location transparency by encapsulating the message-passing details and provides programmers an interface that allows them to migrate a monolithic program into a distributed setting, while preserving the semantics and without massive changes to the syntax of the program. This study provides a perspective on designing distributed systems: A monolithic program can be viewed as an abstraction of a distributed program, specifying *what* functionalities a program attempts to achieve instead of *how* these functionalities are achieved in a distributed setting by abstracting away the details of distributed memory management and message passing over a network.

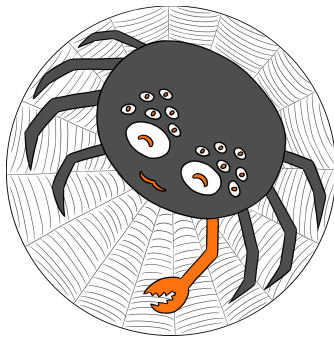
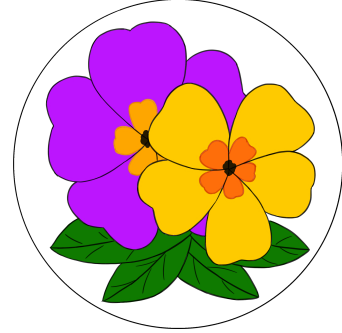
The third conceptual question is: How do we characterise the relationship between the syntax and semantics of programming languages? The study, “*Shoggoth: A Formal Foundation for Strategic Rewriting*”, addresses this question by giving three different models of formal semantics: denotational semantics, big-step operational semantics, and axiomatic semantics, to a core calculus of strategic rewriting languages which is used for composing syntactic transformations. The study provides a per-

spective on the relationship in the context of rewriting: The syntax and semantics are interdependent, since we have observed that transformations of the syntax of expressions encode the meaning for the evaluation of these expressions. In the meanwhile we can characterise and reason about executions of compositions of these syntactic transformations of expressions by analysing their semantics.

Lay Summary

This doctoral thesis presents three distinct studies concerning the meaning of computer programs. Each of these studies addresses a key conceptual question related to programming language design.

Specification and Implementation: The first study investigates how programmers can more effectively articulate what they want a computer to do (specifications) rather than how it should do it (implementations). We introduce Primrose, a tool that allows programmers to define what a container should accomplish without being concerned about how it achieves it. This separation enables more efficient and automated selection of container data types in software development.



Monolithic and Distributed Programs: The second study explores how to understand and manage distributed programs using the same principles as monolithic programs. we present the Universal Method Invocation (UMI) library for the Rust programming language. UMI simplifies the transition from monolithic to distributed programs by concealing the complexities of message-passing and distributed memory management, thereby allowing programmers to maintain the same program structure and functionality across different computing environments.

Syntax and Semantics: The third study examines the relationship between the structure (syntax) and meaning (semantics) of programming languages. We introduce Shoggoth, a formal framework of strategic rewriting in programming languages. This study demonstrates that the way expressions are written (syntax) and their meaning (semantics) are closely interlinked, and understanding this relationship is crucial for accurately predicting and reasoning about the behaviour of programs.



In summary, these studies provide insights into designing effective programming languages and frameworks, emphasising the importance of formal specifications, seamless transitions between different program architectures, and the interplay between syntax and semantics.

Acknowledgements

First of all, I would like to thank my supervisor Michel Steuwer. Having Michel as my supervisor is one of the best things could ever happen in my life. I started working with Michel when I was an undergraduate student in University of Glasgow. During the five years that we worked together, Michel provided unlimited freedom, trust, and support for me to explore areas that I am curious about. Because of his guidance, I am able to keep questioning what the meaning of the world is and pursuing what I *really* want for my life.

I had a great time being a member of Michel’s ComPL research group. I would like to express my gratitude to everyone I met in this group, including Bastian Köpcke, Martin Lücke, Johannes Lenfers, Rongxiao Fu (傅荣梏), Thomas Koehler, Federico Pizzuti, Rudi Schneider, Nicole Heinmann, and Selkan Muhcu.

I sincerely thank Tianyi Li (李天翼) and Leyang Xue (薛乐阳) for being my supportive and caring friends. Particularly, I would like to thank Tianyi, who studies computational linguistics, for his vitally important contributions to my understanding of the conceptual correspondence and theoretical connections between formalism and analysis of natural languages and programming languages. I would like to thank Leyang, for providing me delicious food and taking care of my pet Medusa, which is a Venus flytrap.

I sincerely thank my collaborator Rob van Glabbeek for all the inspiring discussions. In addition, I will always remember the question he asked me when I was having a severe life crisis and considering giving up: “Do you want to be a scientist?” My answer was yes, and my answer is still yes.

Thanks to Tobias Grosser for always being helpful and supportive. Especially, I would like thank him for inviting me to visit his group in University of Cambridge. I spent a wonderful week with them during my stay in Cambridge.

Thanks to my second supervisor Sam Lindley for his insightful inputs in my annual reviews, thesis writing, and career decision making, as well as his apples.

Thanks to all researchers and my collaborators who have positively contributed to my research, including Ohad Kammar, Peter Höfner, Glynn Winskel, and Dan Ghica.

Thanks to everyone in the Friday standup group, especially Wenhao Tang (唐雯豪) and Anton Lorenzen. Thanks to everyone in my office IF 2.33.

I would like to thank my parents for being supportive to all life decisions I have made, and for their financial support allowing me to build a safe base that I could call

a “home” in Edinburgh.

I would like to thank my friend Ke Shen (申可), who is one of the most talented young researchers and brave people I have met. Although there are a lot of feelings that cannot be expressed by words, I do hope you, as a wonderful researcher, to keep pursuing your dream.

I would like to thank my most special friend Yichen Xu (徐逸辰). Thank you for approaching and visiting the glass tower that I live in and I am trapped in. And thank you for agreeing to accompany me to step out of the tower, and to live a life.

I would like to thank my therapist Paulina Nowak. During the past two years, we have worked closely together to understand my post-traumatic stress disorder (PTSD). Because of you, I am able to be aware of and step out of the vivid past that I was trapped into, and look forward.

This thesis is assembled during my visit to different universities, including EPFL, TU Berlin, and University of Cambridge, as well as different cities, including Lausanne, Geneva, Montreux, Paris, Berlin, Cambridge, Vienna, and Budapest. I would like to thank everyone I met during the journey.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. This thesis contains published papers, specifically:

- Chapter 2 is based on the paper: Xueying Qin, Liam O'Connor, and Michel Steuwer (2023). Primrose: Selecting Container Data Types by Their Properties. Art Sci. Eng. Program., 7(3)
- Chapter 4 is based on the paper: Xueying Qin, Liam O'Connor, Rob van Glabbeek, Peter Höfner, Ohad Kammar, and Michel Steuwer (2024). Shoggoth: A Formal Foundation for Strategic Rewriting. Proc. ACM Program. Lang., 8(POPL)

I declare that as the first author of both papers, I made most significant contributions amongst all my co-authors, including proposing these projects, designing and implementing these frameworks, conducting experiments and case studies, major formalisation work, writing papers, and preparing artifacts.

Xueying Qin
秦雪莹

Table of Contents

1	Climbing the Tower of Babel	
	<i>Introduction</i>	1
2	Specifications, All Too Specific	
	<i>Selecting Container Data Types by Their Properties</i>	15
2.1	Introduction	16
2.2	Motivation	18
2.3	Overview	21
2.4	Property Specifications	23
2.4.1	Syntactic Properties as Traits	25
2.4.2	Semantic Properties as Predicates	25
2.4.3	The Interaction between Semantic Properties and Syntactic Properties	27
2.5	Library Specifications	28
2.5.1	The Basic Design of Library Specifications	29
2.5.2	The Library Specification of A LinkedList	31
2.5.3	The Library Specification of A BTreeSet	33
2.5.4	The Library Specification of A HashSet	35
2.5.5	Abstracting Over Implementation Details with Library Specifications	36
2.6	Selecting and Ranking Implementations	36
2.6.1	Selecting Container Implementations Satisfying Syntactic Prop- erties	36
2.6.2	Selecting Container Implementations Satisfying Semantic Prop- erties	37
2.6.3	Handling Interactions Between Semantic and Syntactic Prop- erties	39

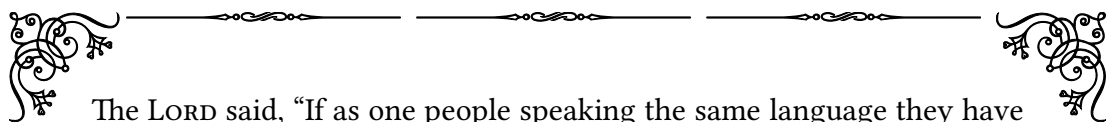
2.6.4	Code Generation and Ranking Implementations by Performance	40
2.7	Evaluation	41
2.7.1	Correctness of Container Implementations w.r.t Their Library Specifications	41
2.7.2	Evaluation of Primrose’s Selection Time	42
2.8	Discussion of Limitations	43
2.9	Related Work	43
2.10	Conclusion	45
3	Oxidising Remote Procedure Calls	
	<i>A Universal Method Invocation Library for Rust</i>	49
3.1	Introduction	50
3.1.1	Contributions	50
3.1.2	Limitations	51
3.2	Background	52
3.2.1	Remote Procedure Calls	52
3.2.2	Rust	53
3.3	The Implementation of the Rust UMI Library	54
3.3.1	Overview	54
3.3.2	The Design of the Translation	55
3.3.3	Resource Management	56
3.3.4	Passing Remote Invocations via Messages	57
3.3.5	Extending Borrow Checking into Distributed Settings	58
3.3.6	Extending Lifetime Management into Distributed Settings	62
3.4	The Operational Semantics	63
3.4.1	The Revised Syntax and Semantics of FR	64
3.4.2	The Syntax and Semantics of dFR	67
3.4.3	Preservation of Semantics when Translating a FR Program into a dFR Program	72
3.4.4	Summary	80
3.5	Related Work	80
3.6	Conclusion	83
3.A	The Type System of FR	85
3.A.1	Preliminaries	85
3.A.2	Typing Rules	88

4	Capturing A Shape-Shifter: The Semantic Process	
	<i>A Formal Foundation for Strategic Rewriting</i>	91
4.1	Introduction	92
4.2	The Syntax of System S	95
4.3	The Semantics of System S	97
4.3.1	The Plotkin Powerdomain	97
4.3.2	Formalised Denotational Semantics	99
4.3.3	Formalised Big-Step Operational Semantics	104
4.3.4	The Denotational Semantics is Equivalent to The Big-Step Operational Semantics	107
4.4	Location-Based Weakest Precondition Calculus	110
4.4.1	Modelling Traversals	112
4.4.2	The Calculus	112
4.4.3	The Soundness of the Weakest Precondition Calculus w.r.t. the Formal Semantics	117
4.5	Reasoning About Strategies with Weakest Precondition Calculus	119
4.5.1	Reasoning About Termination	120
4.5.2	Reasoning About Well Composed Strategies	121
4.5.3	Reasoning About Beta-Eta Normalisation	122
4.5.4	Discussion	125
4.6	Related Work	126
4.7	Conclusion and Future Work	129
5	The Thorn and The Bird: Still We Do It	
	<i>Conclusion</i>	133
	Bibliography	137

Chapter 1

Climbing the Tower of Babel

Introduction



The LORD said, “If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other.”

So the LORD scattered them from there over all the earth, and they stopped building the city. That is why it was called *Babel* — because there the LORD *confused* the language of the whole world. From there the LORD scattered them over the face of the whole earth.

— Genesis 11:1–9 NRSVUE



HUMANS are divided into different linguistic groups, therefore, they are unable to understand each other. That said, “meaning” is encoded into different conceptual schemes in different languages; without accurate translations, it is ultimately difficult for people who speak different languages to communicate with each others. In addition, due to the nature of natural language being ambiguous, it is hard to accurately reason about what others *really mean* even when they speak the same language.

Over the decades, there have been various studies concerning the *meaning* of language. The term *semantics* is used to refer to the studies of linguistic meaning (Katz, 1972; Palmer, 1981). From a philosophical perspective, there are different theories

of meaning. Lewis (1970) describes two topics of the studies of meaning. The first topic corresponds to the first observation from the mythology – “meaning is encoded into different conceptual schemes in different languages”. This topic concerning the meaning of languages is to understand the psychological and sociological facts that a person or a group of people give certain meanings to the symbols in their languages (Lewis, 1970). One kind of approaches are *ideational theories* (Chapman and Routledge, 2009). These theories examine the meaning in terms of and as an output of people’s mental representations (Stich and Warfield, 1994). A different point of view is initiated by Kripke (1980), who argues against the idea of proper name being synonymous with definite descriptions, while proposing that names are associated with their referents through a causal chain of reference. Such a *causal theory* further suggests that the meaning of an expression instead of being inherited from mental states, is determined by the causal connections that the expression has with the objects or concepts that it refers to.

The second topic corresponds to the second observation – “it is hard to accurately reason about what others really mean even under the context of a same language”. This topic is to accurately examine and analyse the meaning of an expression (i.e., a word or a sentence) in a given language (Lewis, 1970). Specifically, Frege (1892) introduces a *theory of reference*, suggesting that meaning of an expression involves both its reference to an object, which is a proper name that contributes to the truth value of a sentence, and its sense, which is how the object is presented. Using the example sentence “the present King of France is bald”, *Russell’s theory of description* (Russell, 1905) argues that Frege’s notion of sense and reference is not sufficient for analysing an expression which has sense but no reference, while introducing a rigorous analytic method for problematic propositions, concerning denoting phrases, making use of the machinery of first-order logic featuring propositional functions. Following Tarski’s (1944) truth definition of a sentence, Davidson (1967) proposes an approach with the core idea that meaning should be understood based on a *formal theory of truth*. There are also *semantic internalism* theories (Mcgilvray, 1998; Chomsky, 2000; Pietroski, 2017) that instead of giving truth value to expressions, view the meaning of an expression as what is used for building a particular mental representation. Taking a holistic approach to analyse the meaning of expressions, *inferential semantics* theories (Brandom, 2000) argue against the idea of using established truth conditions to further analyse good and bad inferences. Instead, these theories suggest to first study the distinction between good and bad inferences, which provides the basis for under-

standing truth conditions, hence the meaning of an expression is studied in relation to other expressions.



Linguists tend to adopt less abstract approaches to analyse the meaning of languages. There are also many different topics within linguistic studies of semantics.

One important field of linguistic semantics is *lexical semantics*, which concerns the meaning of words (Palmer, 1981; Pustejovsky, 2006; Geeraerts, 2017), including topics such as the *semantic structure* of words like ambiguity and polysemy as well as the semantic relations between words such as metaphor and metonymy. In particular, *lexical fields* (alternatively *semantic fields*), which were initially introduced by Trier (1931), study the meaning of words according to their relationship to other words of which the meanings are interdependent (Palmer, 1981; Jackson and Amvela, 2000), and *lexical relations* study the structural relation between words like synonymy and antonymy (Geeraerts, 2017).

Another widely explored field of linguistic semantics is *structural semantics*, or more general, *structural linguistics*, which is inspired by de Saussure's (1916) semiotic analysis centring linguistic signs, attempting to analyse a language as a structured system of interrelated elements. In particular, the aforementioned topics like semantic fields and lexical relations as well as other semantic relations between words have been taken from the lexical studies and further developed into a structured basis for the analysis of words' meaning (Geeraerts, 2017). Such an influential approach has later been adapted into the studies of generative grammars and their formalism (Katz and Fodor, 1963; Chomsky, 1975).

Cognitive semantics is a major part of cognitive linguistics, which is an important linguistic field that has been explored for decades by Johnson (1987), Lakoff and Johnson (1980), Langacker (1987), Fauconnier and Turner (1998). Taking an alternative approach from the structural linguistics, it studies the meaning of languages under the context of human cognition, specifically, viewing semantics as a conceptual organisation of languages (Talmy, 2000; Croft and Cruse, 2004).

Perhaps along with the emerging natural language processing technology such as ChatGPT, the field *computational linguistics*, concerning modelling natural languages as computational models, has now become one of the most popular areas. *Computational semantics*, as an important study within the scope of computational linguistics, enables automatic analysis of sentences' meaning via machines (Mitkov, 2022). Based on the aforementioned fields such as lexical semantics and structural se-

mantics as well as formal semantics like Montague semantics (Montague, 1970) that will be later discussed, the main focus of this field is the *representation* of meaning, where the meaning of the languages are represented via some formal structures, for instance, logic forms including propositional logic (Boole, 1854) and first order predicate logic (Frege, 1879), discourse representation structure studies in discourse representation theory (Kamp and Reyle, 1993), and event structures (Pustejovsky, 1991), tense logic (Prior, 1955; Kamp, 1968) as well as the temporal anaphora that provides representations of events and time in sentences (Partee, 1984; Hinrichs, 1986). Semantic parsing is the technique being studied to transform sentences into these semantic representations, while semantic analysis and semantic inference are performed on these semantic representations for automatically processing the meaning of sentences.

Originated from philosophical view, especially the logic of languages, linguistic formal semantics focusing on analysing the truth condition aspect of meaning with frameworks concerning compositionality, specifically, a formal analysis of the semantics of some language is achieved based on a syntactic formalism of a language (Portner and Partee, 2002). Although obviously the study of linguistic formal semantics does not intend to explore all features of the semantics of the natural languages, it explores ways to precisely model of the syntax as well as analysis of semantics of sentences. An important work within such a field is categorial grammar, of which Ad-jukiewicz's (1935) development provides a formal syntactic approach for analysing higher-order logic. Such an approach has been later adapted for analysing natural languages, such as Lambek calculus (Lambek, 1958). Montague semantics is a model-theoretical approach, providing a relation between syntax and semantics (Montague, 1970). It analyses a subset of English, taking the form of Montague grammar, with lambda calculus, higher-order functions, and type theory. Combinatorial categorial grammar (Steedman, 2001; Steedman and Baldridge, 2011) further provides formalism of categorial grammar with combinatorial logic, which shares the same level of expressiveness as lambda calculus.



As we have discussed in the previous paragraphs, within the scope of linguistic formal studies concerning structured and systematic analysis of the meaning of languages including computational and formal semantics, various formal systems and logic frameworks such as lambda calculus, first order logic, modal logic, combinatorial logic, category theory etc. are used in modelling formal syntactical representations of natural languages to enable precise analysis of the semantics as well as

to facilitate machines to understand and generate meaningful structured sentences. Perhaps not surprisingly, these formal systems and logic components also form an important foundation for the design and analysis of *programming languages*.

Programming languages are created by humans, serving as an interface for humans to communicate and interact with computers. Since humans tend to encode and express information in terms of structured phrases and sentences, like natural languages, programming languages are designed to have grammar and syntax for humans to convey their intentions to computers taking the form of *computer programs*. While computers execute binary code containing zeros and ones, the communication processes between humans and computers are facilitated by compilers, which are responsible for translating information encoded by humans taking the form of computer programs into executable machine code. The study of programming languages explores different approaches for effectively expressing better abstractions in various application domains, facilitating an accurate and efficient compilation process, and providing better frameworks for humans to understand and reason about the behaviours of computers' executions of programs.

The study of formal semantics of programming languages has been around for decades and three main forms of formal semantics have been used serving as a foundation for understanding and reasoning about computer programs (Winskel, 1993; Pierce, 2002). Specifically, *operational semantics* provides meanings to programs by modelling how computations get executed. In particular, *small-step operational semantics* focuses on the incremental reduction of expressions or states, providing a detailed and precise understanding of program behaviours, while *big-step operational semantics* describes the execution of programs in terms of its overall behaviours or outcomes rather than its individual intermediate execution steps. Operational semantics is particularly useful for the implementation of a programming language. *Denotational semantics* gives meanings to programs by modelling the result of computations as mathematical objects. It abstracts away the details of the execution of programs and gives an elegant mathematical model presenting the core concepts of a programming language. Instead of modelling how computations get executed or what are produced by executions of computations, *axiomatic semantics* provides meanings to programs by specifying properties satisfied by the results produced by executions of computations. In practice, it is particularly useful for building a proof system for reasoning about the execution of programs.

In my three-year short research journey, my fundamental motivations are to gain

precise understanding of humans' mental models of computer programs and to improve the design of programming languages in order to allow humans to effectively communicate with computers. Corresponding to the foundational programming languages semantics study, I set out to explore ways to design better abstractions to *model* and *understand* programs in order to allow programmers to effectively *communicate* what they want computers to do in terms of programs and precise formal frameworks allowing us to *reason about* the behaviours of complicated realistic programs. At the end of this journey, I have asked three conceptual questions, and conducted three studies concerning the meaning of computer programs, utilising modelling and reasoning techniques presented in the existing studies of formal semantics of programming languages.



The first conceptual question asked is: How to design better abstraction mechanisms that allow programmers to effectively express *what* they want a computer to do via some declarative yet accurate *specifications* instead of *how* a computer should accomplish a task via some concrete *implementations*?

Formal specification is a rigorous and systematic approach to defining the behaviour, structure, and properties of a system with a specification language of which the syntax and semantics are formally defined such as Z (Spivey, 1989), Alloy (Jackson, 2006), and VDM (Jones, 1990) or logics like first-order logic, linear temporal logic (LTL), and computational tree logic (CTL). It involves describing a system's functionality, constraints, and requirements using precise and unambiguous terms that can be understood both by humans and potentially processed by automated tools for verification and validation.

The main application of formal specifications is to verify and validate software systems, ensuring desired properties are satisfied. In addition, formal specifications of a system also form a clear and precise documentation of the system's requirements, which facilitate communication amongst developers and maintainers of the system.

We make use of another application of formal specifications to address this conceptual question, which is *abstraction*. Formal specifications allow developers to focus on high-level requirements instead of detailed implementations of a system. Such an abstraction not only facilitates managing the complexity in software development but also opens up opportunities for achieving better automation and optimisation in the process of implementing a software system. In the first project, we studied

container types in programming languages and their properties, focusing on declaratively describing the properties of container types using formal specifications rather than having these properties concretely implemented for the container types.

Container data types are ubiquitous in computer programming, enabling developers to efficiently store and process collections of data with an easy-to-use programming interface. Many programming languages offer a variety of container implementations in their standard libraries based on data structures offering different capabilities and performance characteristics. However, choosing the *best* container for an application is not always straightforward, as performance characteristics can change drastically in different scenarios, and as real-world performance is not always correlated to theoretical complexity. Based on this observation, we bring up a research question: How to design a notion of a container that truly allows to separate its interface and usage from the implementation and to infer the implementation from its interface and usage?

This question is answered by the project — *Primrose: Selecting Container Data Types by Their Properties*. In this project, we present Primrose, a language-agnostic tool for selecting the best performing valid container implementation from a set of container data types that satisfy *properties* specified by application developers. Primrose automatically selects the set of valid container implementations for which *library specifications*, written by the developers of container libraries, satisfies the specified properties. Finally, Primrose ranks the valid library implementations based on their runtime performance. With Primrose, application developers can specify the expected behaviour of a container as a type refinement with *semantic properties*, e.g., if the container should only contain unique values (such as a *set*) or should satisfy the LIFO property of a *stack*. Semantic properties nicely complement *syntactic properties* (i.e., traits, interfaces, or type classes), together allowing developers to specify a container’s programming *interface* and *behaviour* without committing to a concrete implementation. We present our prototype implementation of Primrose that preprocesses annotated Rust code, selects valid container implementations and ranks them on their performance. The design of Primrose is, however, language-agnostic, and is easy to integrate into other programming languages that support container data types and traits, interfaces, or type classes. Our implementation encodes properties and library specifications into verification conditions in Rosette, an interface for SMT solvers, which determines the set of valid container implementations. We evaluate Primrose by specifying several container implementations, and measuring the time

taken to select valid implementations for various combinations of properties with the solver. We automatically validate that container implementations conform to their library specifications via property-based testing. This work provides a novel approach to bring abstract modelling and specification of container types directly into the programmer's workflow. Instead of having to select concrete container implementations, application programmers now work on the level of specifications, merely stating the behaviours they require from their container types, and the best implementation is selected automatically. In chapter 2, we discuss this project in detail.

Back to the conceptual question regarding designing better abstraction mechanisms which free programmer from having to choose concrete implementations when writing programs, in this small study, we have demonstrated that property specifications and library specifications describing *what* properties, especially properties giving an account of functional requirements, a container type and its operations should satisfy, which are separated from concrete container implementations describing *how* properties are satisfied. Such separation of concerns enables automation and optimisation for application developers when using a container data type in their programs.



The second conceptual question asked is: How to intuitively understand *distributed programs* using the same conceptual model as *monolithic programs*?

In software architecture design, a monolithic system features a single unit to be deployed (Taylor et al., 2009). Since its components and services are tightly coupled and interconnected, a failure in any one part of it can potentially bring down the entire system, and it scales as a single unit as all components must scale together, even if only one part of the application is experiencing increased load, a monolithic system is inflexible, more susceptible to failures, and less scalable. However, it is easy to implement, deploy, and test, especially for the development of a light-weight service. A distributed system on the other hand contains multiple independent components that can be deployed on different machines, and can communicate and coordinate with each other over a network. A distributed system is often used in large-scale applications and services as although it is more scalable and fault-tolerant, however, it is also more complex to develop, deploy and manage.

There are two motivations to think about conceptually modelling distributed programs as monolithic programs, despite they have very different underlying architectures, design, and implementation. Firstly, it is common to start implementing a system with a monolithic architecture and later migrate it to a distributed design

once the system needs to be expanded to a larger scale. However, migrating a monolithic system to a distributed design usually requires non-trivial effort in changing the program logic and massive re-coding. It would make the migrating process more straightforward if we have a library with which distributed programs can be encoded in the same program logic as monolithic programs. Secondly, the implementation of a distributed system is complicated by the communication between different nodes via message passing as well as locating, querying, and managing resources over a network. It would simplify the process of implementing a distributed system if an application programmer is able to describe the functionality of a distributed program as a monolithic program while message passing and distributed resource management are handled internally by a library or framework.

With such motivations, in the second project, a remote procedural call library for Rust is designed, allowing monolithic programs to be migrated into a distributed setting without massive re-coding, the library automatically extends Rust’s memory safety guarantees into the distributed setting.

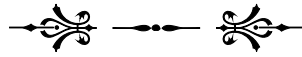
In distributed computing, a remote procedure call (RPC) allows a method invocation to be executed on another computer on a shared network. Such a remote method invocation has the same coding as a local method invocation, without the programmer explicitly encoding the details for the remote interaction. However, it is hard to support *location transparency*, i.e., in existing RPC frameworks such as Java RMI and Rust tarpc, remote method invocations do not have *the same semantics* as local method invocations. In addition, *memory management* is hard in a distributed setting, for instance, distributed garbage collection is known to be complicated.

To address these issues, we design a *universal method invocation* (UMI) library in Rust supporting location transparency. With the UMI library, syntactically, a distributed program is written (almost) the same as its monolithic counterpart; semantically, a distributed program preserves the semantics of a monolithic program. We choose Rust as the target language for the UMI framework to utilise Rust’s memory safety guarantees. Rust is a high-level system programming language which *guarantees memory safety* and *prevents data races* by its *ownership and borrow checking system* for memory management and checking object lifetime of all references in a program during compilation. Since Rust has semantics that guarantees memory safety, we can extend such guarantees to the distributed computing setting, allowing our UMI framework to provide safe remote method invocations.

We provide a usable *Rust implementation* of the UMI framework and formalise

the *small-step operational semantics* for a core calculus of monolithic and distributed Rust programs. In addition, we prove a *location transparency theorem*: When a monolithic program is deployed to multiple nodes with the UMI framework, its semantics is preserved. This project is discussed in detail in chapter 3.

Back to the conceptual question regarding to understanding distributed programs using the same conceptual model as monolithic programs, this project provides a perspective to answer this question: A monolithic program can be viewed as an abstraction of a distributed program, specifying only *what* functionalities a program attempts to achieve instead of *how* these functionalities are achieved in a distributed setting by abstracting away the details of distributed memory management and message passing over a network. Such a perspective, in addition to the technical contributions provided by this project, is conceptually meaningful in modelling and designing distributed systems.



The third conceptual question asked is: How do we characterise the relationship between the *syntax* and *semantics* of programming languages?

I am certainly not the first person asking such a question. In fact, in the world of linguistic studies, the “linguistics wars” (Newmeyer, 1986) happened in 60s and 70s were a academic dispute on the relationship between the syntax and semantics of natural languages. Dating back to the 50s, by presenting the sentence “Colourless green ideas sleep furiously”, which is grammatically correct but nonsensical, Chomsky (1957) argues that the syntax of a language is independent from the semantics. However, some structural linguists emphasise that the analysis of language structure and meaning should be within a synchronic framework.

In programming languages studies, it is generally agreed that the syntax, which represents the form of programs, organises the symbols and defines the programs’ structure without giving the meaning to the programs. After defining the syntax, the semantics, which is the meaning, is then assigned to syntactically valid programs, by formally describing the execution of programs.

While the syntax and semantics concern different aspects of the design of programming languages, they are not completely independent. In the two previous studies, we have already discussed designs concerning syntactic properties of collections of data and minimising the changes in syntax while changing the architecture of programs. There are some conceptually important observations of the design of the syntactic constructs.

Although comparing to modelling semantic properties and reasoning about the semantic preservation, these syntactic constructs seems to be too straightforward to discuss in detail and it is not straightforward to evaluate how “well” they have been designed, they are still very important serving as an abstraction that allows and facilitates programmers to convey the intend semantics of programs. In Primrose, we have seen that some semantic properties like LIFO that depend on syntactic properties, giving a characterisation of a container by stating the behaviours of specific operations given by syntactic properties. In the design of the UMI framework, the syntax serves as the abstraction for location transparency, where the location of resources is abstracted away from how programmers interact with them, while the semantics gives the meaning, for instance, remote memory allocation and remote borrowing, to such an abstraction. Utilising the semantics we are able to reason about that resources are properly used and managed independent from where they are stored.

In the third project, we further studied the syntax and semantics of programming languages from yet another perspective, via rewriting. Rewriting is a versatile and powerful technique used in many domains including symbolic computation, theorem proving, programming language semantics, and compiler optimisation. While being practically useful, rewriting is also conceptually intriguing. In a rewriting system, syntactic transformations are used to systematically encode the semantics of the reduction, simplification and evaluation of expressions. Since these syntactic transformation steps are composable, a valid composition of valid syntactic transformation steps forms a meaningful program, the process of composing syntactic transformation steps together has its own rich semantics.

In practice, *strategic rewriting* is a systematic technique that allows programmers to control the application of rewrite rules by composing individual rewrite rules into complex rewrite strategies. While these strategies have concise and intuitive syntactic constructs and simply serve as compositions of syntactic transformations, they are semantically complex, as they may be nondeterministic, they may raise errors that trigger backtracking, and they may not terminate. Given such semantic complexity, it is necessary to establish a formal understanding of rewrite strategies and to enable reasoning about them in order to answer questions such as: How do we characterise errors and divergence in a strategic rewriting system? How do we understand and model nondeterminism in the executions of strategies? How do we know that a rewrite strategy terminates? How do we know that a rewrite strategy does not fail because we compose two incompatible rewrites? How do we know that

a desired property holds after applying a rewrite strategy?

These questions are answered by the project — *Shoggoth: A Formal Foundation for Strategic Rewriting*. It provides a semantic foundation for understanding, analysing and reasoning about strategic rewriting that is capable of answering these questions. We provide a denotational semantics of System S, which is a core calculus of strategic rewriting languages like Stratego (Visser et al., 1998; Visser, 2001), Elevate (Hagedorn et al., 2023, 2020), and Strafunski (Lämmel and Visser, 2002), and prove its equivalence to our big-step operational semantics, which extends existing work by explicitly accounting for divergence. We further define a *location-based weakest precondition calculus*, which can be seen as an axiomatic semantics of System S (Visser and Benaissa, 1998), to enable formal reasoning about rewriting strategies. We prove this calculus is sound with respect to the denotational semantics and show how it can be used in practice to reason about properties of rewriting strategies, including termination, that strategies are well-composed, and that desired postconditions hold. The semantics and calculus are formalised in Isabelle/HOL and all proofs are mechanised. This project is discussed in detail in chapter 4.

Back to the relationship between syntax and semantics of programming languages, in this study, the syntax and semantics are interdependent, since we have observed that transformations of the syntax of expressions encode the meaning for the evaluation of these expressions. In the meanwhile we can characterise and reason about executions of compositions of these syntactic transformations of expressions by analysing their semantics. Such an observation, aside of the practical usefulness of the framework Shoggoth which we build to enable formal understanding and reasoning of strategic rewriting languages, is conceptually intriguing as a perspective of the study of the syntax and semantics in the design of programming languages.



This thesis contains both published and unpublished work, a structure of this thesis is summarised here:

- Chapter 2 is based on the paper *Primrose: Selecting Container Data Types by Their Properties*, which is published at *The Art, Science, and Engineering of Programming, Volume 7*;
- Chapter 3 is based on my unpublished work conducted during my internship at Huawei R&D;

- Chapter 4 is based on the paper *Shoggoth: A Formal Foundation for Strategic Rewriting*, which is published at *Proceedings of the ACM on Programming Languages*, Volume 8, Issue POPL;
- Chapter 5 is the conclusion of this thesis.

Chapter 2

Specifications, All Too Specific

Selecting Container Data Types by Their Properties

‘Then you should say what you mean’, the March Hare went on.
‘I do’, Alice hastily replied; ‘at least — at least I mean what I say — that’s the same thing, you know.’
‘Not the same thing a bit’, said the Hatter.
— Lewis Carroll “Alice’s Adventures in Wonderland”

Prologue

PEOPLE often try to *say* what they *mean*, however, there is always a gap between what they say and what they really mean — *they do not mean what they say*. Likewise, programmers often try to *encode* what they *meant* for a computer to do, however, there is always a gap between what they encode and what they really want a computer to execute — *they do not mean what they encode*.

One observation is that in an implementation of an application, what an application developer encode tends to be “all too specific” comparing to what they have modelled in their mind. Take a container type example which is illustrated in later sections, when an application developer wants to use a *unique container* in an application, the application developer *means* that the container does not contain any duplicated elements. However, the application developer often has to *encode* such a required container as a concrete data structure, such as a tree, a hash table, or a

list without duplicated element etc. By choosing a concrete encoding, application developers no longer just express what they mean, but additionally commit to certain performance characteristics or memory consumption which may or may not be desirable for their applications.

Hence, in this chapter, we design a programming framework facilitating application developers to better express what they mean for a computer to execute by writing declarative specifications instead of concrete and fixed implementations. We believe that such a framework enables better automation and optimisation as it gives opportunities to a tool, such as a compiler, to select or generate the best performing implementation according to the specification provided by an application developer.

2.1 Introduction

Container data types, such as sets, lists, and trees, represent collections of data ubiquitous in everyday programming (Cormen et al., 2009). Virtually all programming languages provide a variety of container implementations in their standard libraries.

Much work has been done to design better abstractions, improve performance and verify correctness for container data types. However, a crucial problem for application developers using containers still exists: when choosing a container data type, application developers are forced to select a concrete implementation that comes with certain theoretical complexity and practical performance tradeoffs.

For example, consider representing a mathematical set, i.e., where each element should occur at most once. In C++, we must choose between `std::set`, usually implemented as red-black trees (Bayer, 1972), and `std::unordered_set`, implemented as a hash table. The hash-based implementation was added to the C++ standard in 2011, as the C++ standard has strict complexity requirements preventing the ordinary `std::set` to be implemented as the (often faster) hash table. Many blog posts and discussions (Orr, 2019; Wicht, 2012; Cechner, 2014; Edouard, 2020; Ankerl, 2019) report on the performance of various C++ containers, showing the community's interest and the need for external guidance that the language itself does not provide.

In other languages, the situation is similar. Rust provides two container implementations, `HashSet` and `BTreeSet`, expecting application developers to make an explicit choice between them. Scala's complex collection library features abstract interfaces, such as the `Set` trait, abstracting over many implementations such as `HashSet` and `TreeSet`. But when creating an instance of `Set`, a default `HashSet` implementation

is chosen regardless of the suitability of this implementation choice for the usage pattern of the application.

These examples demonstrate a general problem: Application developers are forced to *overspecify*, by having to select a concrete implementation, where we generally would like application developers to be shielded from low-level implementation details. Application developers should primarily care about the *abstract behaviour* of the containers in their application, and not how this is achieved. The compiler, or a dedicated tool, should identify those containers that satisfy their functional requirements, and select the best implementation automatically.

In this chapter, we propose an automated tool: Primrose, which allows application developers to specify the expected behaviours and programming interfaces of containers as *properties*. *Syntactic properties* specify the required programming interface of the container and are expressed as traits of the underlying programming language. *Semantic properties* specify the expected behaviour of the container and are written as logical predicates used as refinements of the container type. Primrose automatically selects the set of valid implementations for which the *library specifications*, written by the library developers as pre- and post-conditions of the container operations, satisfy the specified syntactic and semantic properties using an SMT solver. Finally, Primrose ranks the valid library implementations based on their runtime performance.

To select the best container implementation, firstly, those container implementations which meet the functional requirements of the application developer must be determined, and then those valid container implementations must be evaluated based on non-functional requirements. While Primrose does include functionality for ranking based on benchmarks, the focus of this study is on the first of these two problems. There are many existing sophisticated techniques for selecting based on non-functional requirements, and they are highly complementary with Primrose.

In this work, we apply verification and formal methods techniques, including refinement types, formal library specifications, and SMT solvers, in an innovative way to raise the level of abstraction for developers, freeing them from the burden of choosing container implementations, and opening up the possibility to automatically improve the performance of applications.

To summarise, this study makes the following contributions:

- We present Primrose (section 2.3), a language-agnostic tool for selecting valid container implementations (section 2.6) based on *properties* (section 2.4) used to describe their behaviour and programming interface, and ranking them based

on their performance.

- We show a new application of refinement types not—as previous work did—for verification purposes, but to raise the level of abstraction for developers and to improve the runtime performance of applications with container data types (section 2.4).
- We develop a new methodology to specify container libraries, amenable to our selection process, making use of existing formal methods work such as data abstraction and Hoare logic (section 2.5).
- We show the feasibility of Primrose, selecting container implementations that satisfy various properties from a Rust library of eight container types with library specifications. We validate container implementations against specifications and evaluate the efficiency of the selection process (section 2.7).

2.2 Motivation

Suppose as part of a larger application, we want to find and store all the elements of a larger collection, but without duplicates. We might, for example, use the result of this function to count the number of unique elements or process the elements further, now with the guarantee that each element in the returned collection is unique.

An easy way to implement this is to return a container that only permits unique elements. We might think of a *set*, however, as discussed in section 2.1, this requires a choice: Which concrete implementation of the abstract idea of a mathematical set should we use?

Figure 2.1a shows a Rust code snippet computing a container `uniqueElements` that contains the unique elements of the original `input` sequence. The application developer must choose a concrete container implementation, such as `HashSet` in line 1, but other valid choices would be Rust’s `BTreeSet` (line 2), or perhaps a custom `UniqueVect` (line 3) container, which stores all elements in a vector but ensures there are no duplicates, or some other `FancySetImplementations` (line 4). Whether a container implementation is *valid* is determined by the application developer’s *functional requirements*. Our uniqueness requirement, for example, is not met by the Rust `HashMultiSet` (line 5). If the application developer also required elements to be stored in a particular order, this would also rule out the `HashSet` implementation.

<pre> 1 type Set<I> = HashSet<I>; Rust 2 // type Set<I> = BTreeSet<I>; 3 // type Set<I> = UniqueVect<I>; 4 // type Set<I> = FancySetImpl<I>; 5 // type Set<I> = HashMultiSet<I>; ? 6 7 let mut uniqueElements 8 = Set::new(); 9 for val in input.iter() { 10 uniqueElements.insert(val); }</pre>	<pre> 1 property unique { Primrose 2 \c -> (for-all-elems (\a -> 3 (unique-count? a c)) c) }; 4 type UniqueCon<I> = { 5 c <: ContainerT unique c }; 6 7 let mut uniqueElements Rust 8 = UniqueCon::new(); 9 for val in input.iter() { 10 uniqueElements.insert(val); }</pre>
---	--

(a) In Rust, application developers must choose a concrete container implementation with potentially surprising performance implications.

(b) Using Primrose, developers describe the container's expected behaviour via *properties* and the best valid implementation is selected.

Figure 2.1: Selecting the unique elements of a sequence by inserting the elements into a *set*.

Many programming techniques exist to abstract over multiple concrete implementations of a general concept. In object-oriented languages, *abstract classes* enable hiding multiple implementations behind a common interface. Similar features exist in other languages under different names, such as, *traits* (e.g., in Rust and Scala), *protocols* (e.g., in Swift), *interfaces* (e.g., in Java), and *type classes* (e.g., in Haskell). All these techniques allow developers to use multiple concrete implementations, such as `HashSet` and `BTreeSet`, with a single abstract type, which we might call `Set`. However, these types are deliberately *abstract*, meaning that we *cannot* instantiate them directly: When creating such a type, a developer must commit to a specific concrete implementation, requiring the developer to look underneath layers of abstraction to make an informed decision. Thus, these abstraction techniques do not free developers from considering low level details and they are not powerful enough to express *semantic* requirements: developer cannot specify their functional requirements directly, but merely provide a common *syntax* enabling the use of multiple implementations. With such an abstract container type `Set`, we cannot express that each concrete implementation is required to contain no duplicate elements. Similarly, with an abstract type `Stack`, we cannot state that the last-in-first-out property is respected by the `push` and `pop` operations.

Figure 2.1b shows the same problem of selecting unique elements, but expressed using Primrose. Application developers specify their functional requirements—in this

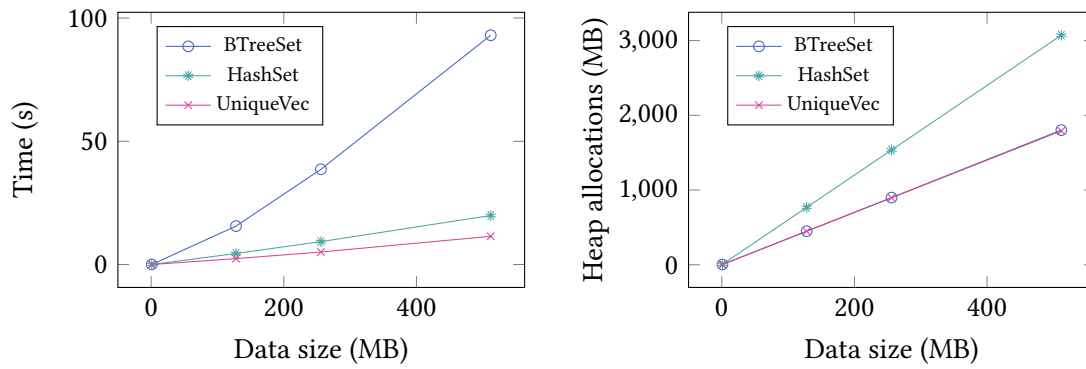


Figure 2.2: Runtime performance (left) and memory consumption (right) of three container implementations for storing unique elements of an input sequence from 2.1a. The custom `UniqueVec` implementation ensures elements to be unique lazily on access. It is the fastest implementation, outperforming `HashSet` and `BTreeSet` from the Rust standard library, while consuming less memory than `HashSet`.

case, that the container must contain unique elements—as a *semantic property*. This semantic property is expressed in lines 1–3 in the Primrose specification language as a logical predicate written as a lambda expression. The property is used to *refine* the container data type in lines 4 and 5. Refinement types have long been used as a technique for program verification—including container types (Vazou et al., 2013). Here, we use refinement types in a new way, allowing programmers to express the expected behaviour of a container, and freeing them from having to make a (potentially difficult) implementation choice. The remaining code remains unchanged: we can simply use the refined type in line 7. Primrose preprocesses the code from figure 2.1b, identifies all valid container implementations from a library of containers, and generates a program equivalent to figure 2.1a with the best container implementation inserted automatically.

However, which is the *best* container implementation? This depends on the non-functional requirements of the application: Often developers care about fast runtime performance, also, for example, an application might require a low memory footprint. Figure 2.2 shows the performance and memory consumption for three different implementation choices. Perhaps surprisingly, a custom `UniqueVec` implementation that uses a vector and lazily ensures that the stored elements are unique, by sorting the vector and removing duplicates on access, outperforms the Rust built-in containers `HashSet` and `BTreeSet`. In addition, it is also the best choice for machines with limited memory. Choosing the best container implementation is not always straightforward, particularly as theoretical complexity of operations can sometimes be mis-

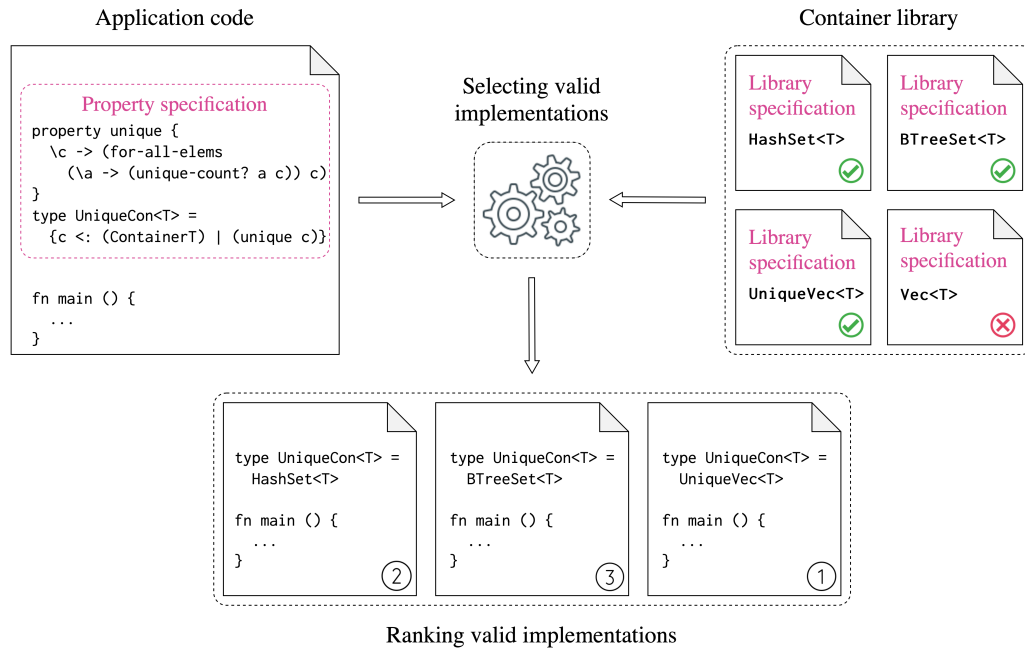


Figure 2.3: The workflow of Primrose: *Property specifications* (top left), written and used by the application developer, are used to check which *library specifications* (top right), written by library developers, satisfy them. Valid implementations (marked with a green check marks), are then ranked by their performance (bottom).

leading in the presence of practical effects such as cache-friendliness. Primrose selects implementations satisfying developers’ functional requirements and opens up opportunities to automatically choose the most desired implementation according to non-functional requirements.

2.3 Overview

Figure 2.3 gives an overview of the design of the Primrose selection tool. Using Primrose, the application developer writes code in terms of an abstract type, and a *property specification* describing the syntactic and semantic properties they expect this type to satisfy. The syntactic properties take the form of traits and the semantic properties take the form of type refinements. To write a program, the developer only specifies what functional properties must be satisfied by the required container, and does not have to commit to a particular implementation. The developer specifies that they require a container (the syntactic property `ContainerT`) where all elements are unique (the semantic property `unique`). We discuss properties in detail in section 2.4.

Given this code as input, Primrose will, acting as a preprocessor, generate copies of

the input code where the abstract type is instantiated into a valid concrete implementation that satisfies the expected properties. It determines which implementations are valid by consulting *library specifications*, which are provided by library developers. These specifications abstract over concrete container implementations and provide a summary of their externally observable semantics. For each implementation, the library specification contains the pre- and post-conditions of each operation in terms of an abstract list model. We discuss these specifications in more detail in section 2.5.

In our example in figure 2.3, the library specification of the `Vec<T>` type indicates that it is not a suitable choice for `UniqueCon<T>`, as it does not satisfy the required semantic property `unique`. We use a satisfiability modulo theories (SMT) solver for the selection process, which we discuss in section 2.6.

Figure 2.3 shows at the bottom a simplified version of the generated programs. Note that in order to make 2.3 concise, we only show a simplified version of the generated programs with selected implementations that does not reflect how traits constrain available operations for interacting with the container type. In our implementation, we ensure that only the container operations that the application developer specifies with syntactic properties are accessible in the generated program. Our current prototype of Primrose focuses on ensuring the functional correctness of selecting container implementations based on desired properties. Nevertheless, we have implemented a simple process that ranks valid implementations by their runtime performance. Rankings by other non-functional metrics could easily be added to our design. We provide discussion about code generation and ranking in section 2.6.4.

Using Rosette as the Common Language for Specifications and Selection The “solver-aided programming language” Rosette (Torlak and Bodík, 2013; Torlak and Bodik, 2014) is used as the common language in Primrose for the formal parts. Rosette is chosen for Primrose due to its convenient interface to the Z3 SMT solver and the straightforward translation from Primrose property specifications into Rosette. Property specifications are used as verification conditions when selecting implementations by checking against library specifications which are directly encoded in Rosette. The selection process is done by interacting with the SMT solver via Rosette.

Portability of Primrose Currently, we choose Rust as the target language to implement our idea. Application developers write the property specifications as a part of their Rust programs and Primrose generates Rust code after processing specifications.

However, Primrose could easily be ported to many other languages, since property specifications, library specifications, and the process of selecting implementations are all language-agnostic and not attached to Rust’s particular type system or language features. Adapting property specifications into other languages only requires such languages to have a construct similar to Rust’s traits, such as traits in Scala and interfaces in Java, allowing us to model syntactic properties. It would be straightforward to add new backends to Primrose to generate code in these languages. Our library specifications are, by design, an abstraction over implementation details, describing the intended semantics of container operations without respect to their implementation. This means we can trivially adapt these specifications to container libraries from other languages, so long as our specifications remain an abstraction of the new implementations. Thus, we anticipate that Primrose could easily be adapted to produce code in any language with sufficient support for data abstraction, such as Java, Scala, Swift, or C++.

2.4 Property Specifications

The application developer specifies the desired behaviours of their required container with a *property specification*, for example, for the type `UniqueCon` from figure 2.3:

```

1  property unique Primrose
2    { \c -> (for-all-elems (\a -> (unique-count? a c)) c) }
3  type UniqueCon<T> = {c <: (ContainerT) | (unique c)}
```

We first define the *semantic property* `unique` using a *predicate*. In our specification language, such predicates have type $Con\langle\tau\rangle \rightarrow Bool$, where $Con\langle\tau\rangle$ is a placeholder that is resolved into a concrete container type by the selection process. The combinator `for-all-elems` is part of a library enabling to write predicates for individual container elements. The predicate `unique-count?` holds if and only if the given element occurs exactly once in the container. These combinators and predicates are explained in section 2.4.2.

With the defined *semantic property* `unique`, we can then declare the container type `UniqueCon<T>`. The first part of the declaration specifies the syntactic property that must be satisfied by the container type, in the form of the trait `ContainerT`. Specifically, `c <: (ContainerT)` says that the type of the container `c` must implement the trait `ContainerT`, which specifies a set of basic container operations. The second part of the declaration *refines* our container type by the predicate `unique`, stating that the prop-

Literals	$l := true \mid false$
Terms	$t := l \mid x \mid \lambda x. t \mid t t$
Refinement	$r := t \mid r \wedge r$
Container Type Declarations	$c := \{v <: B \mid r\}$
Simple Types	$\sigma := Bool \mid T \mid Con\langle\sigma\rangle$
Types	$\tau := \sigma \mid \tau \rightarrow \tau \mid \forall T <: B. \tau$
Bounds	$B := trait_name \mid B, B$

Figure 2.4: The syntax of property specifications. T is the type variable, ranging over element types of the target language, which is Rust in this case.

erty must be invariant across all container operations. Properties may also be composed. For multiple syntactic properties, we specify a list of traits ($c <: (T_1, T_2)$) that the container type implements. For multiple semantic properties, we use conjunction, i.e. $((p_1 \ c) \text{ and } (p_2 \ c))$.

Figure 2.4 shows the syntax of the Primrose property specification language. Formally, the specification language is a variant of the polymorphic λ -calculus (Reynolds, 1974; Girard, 1986), with restrictions on the use of polymorphism to enable implicit type inference (Hindley, 1969; Milner, 1978). This type system guarantees termination, making specifications easier to analyse and straightforward to translate into SMT verification conditions in Rosette. The translation into Rosette is straightforward, as terms in the Primrose property specification language (literals, variables, lambdas, and function application) are translated into their counterparts in the functional Rosette language.

Listing 2.1: The implementation of a container trait

```

1 pub trait ContainerT<T> {
2     fn len(&self) -> usize;
3     fn contains(&self, x: &T) -> bool;
4     fn is_empty(&self) -> bool;
5     fn insert(&mut self, elt: T);
6     fn clear(&mut self);
7     fn remove(&mut self, elt: T) -> Option<T>;}

```

Rust

2.4.1 Syntactic Properties as Traits

In our Primrose prototype, we encode syntactic properties as Rust traits, specifying the operations needed by the application developer to interact with a container. Traits are defined in Rust and lifted into our property specification language. For instance, listing 2.1 shows the trait `ContainerT` introduced above.

By writing `c <: ContainerT`, the application developer indicates that they expect the container type selected by Primrose to include implementations for all operations in the trait `ContainerT`. Thus, after executing Primrose, `UniqueCon<T>` will be resolved into a concrete container type that implements the trait `ContainerT`.

As mentioned, we can also declare a container type that satisfies multiple syntactic properties. For instance, suppose that in addition to `ContainerT`, we would like our container to also satisfy the syntactic property `IndexableT`:

```

1 pub trait IndexableT<T> {                                Rust
2     fn first(&self) -> Option<&T>;
3     fn last(&self) -> Option<&T>;
4     fn nth(&self, n: usize) -> Option<&T>;
5 }
```

With just `ContainerT`, there is no way to observe the *ordering* of elements in the container, but with `IndexableT` there is, as we can now select elements based on their position. By composing our new syntactic property `IndexableT` with `ContainerT` we can now specify a container of unique elements where the order can be observed:

```

1 type UniqueIndexableCon<T> =                               Primrose
2     { c <: (ContainerT, IndexableT) | (unique c) }
```

Semantic properties, such as `unique`, must be invariant across all operations from all syntactic properties required of the container.

2.4.2 Semantic Properties as Predicates

As mentioned, semantic properties are predicates that are used to construct refinements for container types; each declared container type in the form $\{v <: B \mid r\}$ is a *refinement type*, i.e. a type circumscribed by a logical predicate (Freeman and Pfenning, 1991). When the predicates are in SMT-decidable logic, they can be statically checked (Bierman et al., 2010). Such techniques are used in programming languages like Liquid Haskell and F*, where they are used to facilitate verification of program correctness. For instance, in Liquid Haskell, we may define a refinement

type `UniqueList` representing a list of unique elements as:

```

1 {-@ measure unique @-}                                Liquid Haskell
2 unique :: (Ord a) => [a] -> Bool
3 unique [] = True
4 unique (x:xs) = unique xs && not (S.member x (elts xs))
5 {-@ type UniqueList a = {v:[a] | unique v} @-}
```

While our syntax for type refinements strongly resembles Liquid Haskell, our refinement types are slightly different, and serve a different purpose. Firstly, Liquid Haskell’s refinements are attached to a *concrete type*, in this case a list (written `[a]`), whereas our refinements are attached to an abstract container type, which is then resolved by Primrose into a concrete implementation. Secondly, Liquid Haskell uses type refinements for the purpose of *correctness*: If a list is declared to have type `UniqueList`, the Liquid Haskell verifier will check that it satisfies the predicate `unique`. The `notUniqueList` shows that it will report an error at compile time if a given list contains duplicates.

```

1 {-@ notUniqueList :: UniqueList Int @-}                Liquid Haskell
2 notUniqueList :: [Int]
3 notUniqueList = [3, 1, 2, 3]
```

Our work instead uses type refinements to specify the semantic requirements of the application developer to guide selection of valid concrete implementations. Once all valid implementations have been found, Primrose simply selects the implementation providing the best performance for the application developer. In short, rather than to aid verification, we use refinement types to help application developers optimise their programs. We give more details on the selection process in section 2.6.

Combinators and Predicate Functions Demonstrated by our examples, Primrose provides a set of combinators and predicate functions to facilitate writing of property specifications. These combinators and predicate functions are defined in Rosette and then imported into our property specification language. In the semantic property `unique`, the combinator `for-all-elems` is used to specify that the predicate `unique-count?` must hold for all elements inside the container. The type of the combinator `for-all-elems` is $Con\langle\tau\rangle \rightarrow (\tau \rightarrow Bool) \rightarrow Bool$, meaning this combinator takes in two arguments, the first of which is a container and the second of which is a predicate on the elements of that container, and eventually returns a boolean value.

For the purposes of checking, we represent containers $Con\langle\tau\rangle$ abstractly in Rosette as lists. We discuss this list abstraction and justify it in section 2.5. With such a list ab-

straction, we are able to straightforwardly implement our `for-all-elems` combinator with a list fold operation:

```
1 (define (for-all-elems c fn) Rosette
2   (foldl elem-and #t (map (lambda (a) (fn a)) c)))
```

We also provide some combinators for applying *relations* between elements in a container. For instance, the combinator `for-all-consecutive-pairs`:

$$\text{for-all-consecutive-pairs} : \text{Con}\langle\tau\rangle \rightarrow (\tau \rightarrow \tau \rightarrow \text{Bool}) \rightarrow \text{Bool} \quad (2.1)$$

Unlike `for-all-elems`, this combinator gives a binary relation between elements, and checks that this relation holds between any two consecutive elements in a container.

With the combinator `for-all-consecutive-pairs` and the predicates `geq?` and `leq?`, we can define properties like `ascending` and `descending`, which specify particular orderings of elements in a container:

```
1 property ascending { \c -> (for-all-consecutive-pairs c leq?) } Primrose
2 property descending { \c -> (for-all-consecutive-pairs c geq?) }
```

Besides the set of combinators and predicate functions predefined in `Primrose`, application developers may also provide customised functions by providing Rosette definitions and importing them into our property specification language.

Composition of semantic properties As shown in figure 2.4, we can compose semantic properties in a container type declaration with conjunction. For instance, to declare a container type with elements arranged in *strictly* ascending order, i.e., both `unique` and `ascending` properties must hold, we can write the following:

```
1 type StrictlyAscendingCon<T> = Primrose
2   { c <: (ContainerT) | ((unique c) and (ascending c)) }
```

The conjunction `and` is directly translated into a conjunction operation in Rosette.

2.4.3 The Interaction between Semantic Properties and Syntactic Properties

All semantic properties we have seen so far have been invariants across all operations, but some semantic properties relate to specific operations given by syntactic properties. For instance, when specifying a stack container type providing operations `push` and `pop` with the expected last-in-first-out (LIFO) property. Firstly, we define a trait specifying operations `push` and `pop`, namely `StackT`:

Listing 2.2: The trait `StackT` specifying operations `push` and `pop`

```

1 pub trait StackT<T> {                                     Rust
2     fn push(&mut self, elt: T);
3     fn pop(&mut self) -> Option<T>;
4 }

```

Secondly, we define the semantic property `lifo` for containers that implement `StackT`:

Listing 2.3: The semantic property LIFO

```

1 property lifo { \c <: StackT -> (forall \x. pop (push c x) == x) } Primrose

```

Unlike previously, this semantic property includes a requirement that the given container implements the trait `StackT`, enabling us to refer to the operations `pop` and `push` inside the semantic property. In this definition, `forall` is a combinator with type:

$$\text{forall} : \forall x. (x \rightarrow \text{Bool}) \rightarrow \text{Bool} \quad (2.2)$$

This combinator is implemented with the `forall` procedure defined in Rosette’s library, which serves as a construct for creating universally quantified formulae.

Armed with the trait `StackT` and the semantic property `lifo`, we can combine all these elements and declare our stack type as follows:

```

1 type StackCon<T> = {c <: (ContainerT, StackT) | (lifo c)} Primrose

```

In the next section, we will discuss how library developers write specifications for their container implementations.

2.5 Library Specifications

Library specifications abstract over Rust container implementations, providing a clear definition of intended semantics of each operation, without respect to performance or implementation details. This approach allows us to select container implementations by simply checking their library specifications, rather than their full implementations, against the properties specified by the application developer. Moreover, using specifications which are abstracted from implementations makes Primrose easy to repurpose for programming languages other than Rust, as the same specifications would apply, with minimal or no modification, to container libraries written in any other language.

By encoding these specifications into *property based tests*, which validate container implementations against their library specifications (section 2.7.1), we ensure

the selected implementations indeed satisfy a required property specification. Since these library specifications form a *functional correctness* specification for each operation, they could also be used in future as the basis of full functional correctness verification with a verification framework for Rust (Jung et al., 2017a), but this is out of scope for Primrose.

2.5.1 The Basic Design of Library Specifications

Library specifications of concrete container implementations are developed based on Hoare logic (Hoare, 1969a). For each concrete container implementation, we provide a set of *Hoare triples*, one for each operation. A Hoare triple of the form $\{\phi\} \text{ op } \{\psi\}$ states that if the *precondition* ϕ holds and the operation op is executed, then the *post-condition* ψ will hold. These conditions are predicates on the state of the program. In our case, the state contains the container, plus any other inputs and outputs of the operation op .

As mentioned in section 2.3, we model the container as a list in Rosette for Primrose’s library specifications. The list is a model to convey the intended semantics, and does not proscribe anything about the implementation — the implementation is free to represent data in any chosen structure. For example, a set data type may be implemented with a binary search tree, but will still be specified with a list. These model lists are a simple abstraction, easy to analyse, with which all container operations can be specified.

Library Specifications Convey the Intended Semantics for Implementations

It is important that all possible executions of a concrete implementation should be captured by its library specification. Otherwise in the process of selecting implementations by checking if their library specifications match the required semantic property, Primrose could select an unsatisfying implementation. More formally, a proof of functional correctness of an implementation w.r.t. its specification would take the form of a data refinement (de Roever and Engelhardt, 1998), where each value of the concrete container type is related to our list model by an *abstraction function* α , and our specification on lists is shown to contain all possible behaviours of the concrete implementation using a *forward simulation*:

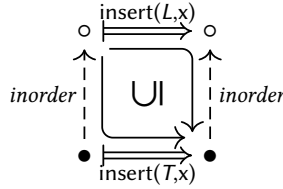
$$\alpha^{-1}; \text{op}(C) \subseteq \text{op}(A); \alpha^{-1}$$

(where ; is forward composition of relations
and α^{-1} is the inverse relation of α)

Here, $\text{op}(C)$ denotes the concrete implementation of our operation op , represented as a relation from inputs to outputs. The abstract operation $\text{op}(A)$ is the maximal relation satisfying the Hoare triple given in our library specification, and α is a suitable abstraction function that flattens a concrete container into a list.

If a forward simulation is shown for all operations, we can then conclude that each possible execution with the concrete container implementation has a corresponding execution with an abstract list, thus the specification accurately captures the implementation's semantics.

For instance, a binary search tree T can be abstracted to a sorted list L by an abstraction function *inorder* that does an in-order traversal. For each operation interacting with T , there exists a corresponding operation at the abstract level defined using L . Take the operation $\text{insert}(T, x)$, which inserts an element x into a binary search tree T . We can abstract such an operation to $\text{insert}(L, x)$ which inserts x at the right location in a sorted list. The relation between these two operations is shown by this diagram:



In this work, we specified four container implementations from Rust's standard library (*Vec*, *LinkedList*, *HashSet*, *BTreeSet*) and four custom container implementations (*SortedVec*, *LazySortedVec*, *UniqueVec*, *LazyUniqueVec*) by abstracting them into a list model. As we discuss in section 2.5.5, library specifications abstract over some implementation details, and, thus, *Vec* and *LinkedList* share the same specifications, as do the eager and lazy *SortedVec* and *UniqueVec* implementations. For each specification, we define a suitable abstraction function for forward simulation which, while not needed for selection, is used for property-based testing to justify that a concrete implementation satisfies the intended semantics described by its library specification.

Completeness of Library Specifications While it is important to ensure that library specifications indeed convey the intended semantics of the implementation,

completeness of library specifications is also important. Without completeness, Primrose could possibly rule out perfectly valid implementations because it cannot prove that the required semantic properties are preserved for an operation of which the specification is incomplete.

Our approach easily ensures completeness when each operation is specified by a *deterministic* model operation. Forward simulation states that every execution of the concrete implementation has a corresponding execution in the abstract operation, while determinism states that such correspondence is one-to-one, i.e., each abstract execution also has a corresponding concrete one. Thus, just as forward simulation states that each property established for an abstract operation applies also (via the inverse of the abstraction function α^{-1}) to a concrete implementation, completeness states that each property established for a concrete implementation applies (via the abstraction function α) to the abstract operation. With both completeness and forward simulation, we ensure that *all* valid implementations and *only* the valid implementations are selected by Primrose.

There are many other available approaches for modelling library specifications, for instance, the axiomatic approach used in algebraic specifications for abstract data types Wirsing (1990), specifying the behaviour of operations as a set of equational axioms that relate various operations. However, it is hard to ensure the completeness of algebraic specifications, as it is hard capture all behaviours of all operations by a set of equations.

2.5.2 The Library Specification of A LinkedList

Rust's `LinkedList` is a doubly-linked list. The abstraction function to convert it into a logic list is straightforward: Collect all nodes' values with previous and next pointers.

Firstly, we specify the insertion operation, `LinkedList::insert`, of which the type signature is:

```
1 fn insert(&mut self, elt: T) {...} Rust
```

Since variables in Rosette are immutable, in the corresponding abstract insertion operation, we alter the type to return a new list instead of altering the list in-place¹:

```
1 abs-insert: List<T> -> T -> List<T> Rosette
```

¹Rosette is untyped, but this is morally the type signature.

We can then provide the specification of `LinkedList::insert` with respect to its corresponding abstract operation, the maximal relation satisfying the Hoare triple:

$$\{xs_0. \text{true}\} \text{abs-insert} \{xs_0 \ x \ xs. \ xs = \text{model-insert } xs_0 \ x\} \quad (2.3)$$

Here, xs_0 refers to the initial value of the container, xs refers to the resultant container, and x is the element we insert. The function `model-insert` is defined in Rosette on lists:

```
1 (define (model-insert xs x) (append xs (list x))) Rosette
```

The postcondition states that we expect applying the insertion operation to a container to produce the same result as the result produced by `model-insert` function. In library specifications, defining such *model operations* is a common technique to simplify writing postconditions.

Similarly, we also provide the specification for the operation `LinkedList::contains`:

```
1 fn contains(&self, x: &T) -> bool {...} Rust
```

In our corresponding abstract operation, in addition to the boolean value indicating whether the given element x is present or not, the input container is also returned, as we would like to express the input container is not mutable, its value remains unchanged after this operation. Also, since the underlying value with type T is given by an immutable reference $\&T$, in the abstract operation we treat the immutable reference $\&T$ as simply T . The signature of the abstract operation is shown below:

Listing 2.4: The signature of the abstract operation corresponding to `LinkedList::contains`

```
1 abs-contains: List<T> -> T -> (List<T>, bool) Rosette
```

The Hoare triple that serves as the specification of `LinkedList::contains` is:

$$\{xs_0. \text{true}\} \text{abs-contains} \{xs_0 \ x \ xs \ r. (xs, r) = \text{model-contains } xs_0 \ x\} \quad (2.4)$$

Note that in this specification, the model operation `model-contains` defined in listing 2.5 has the same type signature as the abstract operation shown in listing 2.4. It also returns a pair of values: the output list, which is always equal to the input list, and a boolean value indicating if the element is present in the list.

Listing 2.5: The model operation for checking an element's containment

```
1 (define (model-contains xs x) Rosette
2   (cond [(list? (member x xs)) (cons xs #t)]
3         [else (cons xs #f)]))
```

Because `model-contains` returns the unchanged list, it specifies that the operation `LinkedList::contains` should not change the list.

The library specification of the list removal operation is slightly more complicated, we use `T?` to denote that a type may be `null` to express Rust's `Option<T>` type, which is the return type of `LinkedList::remove`. The type signature of `LinkedList::remove` is shown below:

```
1 fn remove(&mut self, x: T) -> Option<T> {...} Rust
```

This operation removes the first occurrence of an element from the given linked list and returns it. If the linked list does not contain the element, `None` is returned and the list remains unchanged. The signature of the corresponding abstract operation is:

```
1 abs-remove: List<T> -> T -> (List<T>, T?) Rosette
```

The model removal operation has the same signature as the abstract operation. We return `null` in Rosette for the `None` case:

```
1 (define (model-remove xs x) Rosette
2   (cond [(list? (member x xs)) (cons (remove x xs) x)]
3         [else (cons xs null)]))
```

Again, we return a pair of the resulting list and the element being removed. Then we provide the library specification of `LinkedList::remove`:

$$\{xs_0. \text{true}\} \text{abs-remove } \{xs_0 \ x \ xs \ r. (xs, r) = \text{model-remove } xs_0 \ x\} \quad (2.5)$$

To provide a complete specification of `LinkedList`, the library developer must ensure that each operation of the `LinkedList` is specified by a trait, and for each operation in each trait the `LinkedList` implements, specifications similar to the above are provided.

2.5.3 The Library Specification of A BTreeSet

For the `LinkedList` it is intuitive to use a logic list as a model, as they are both lists. However, even for non-linear structures such as trees, we can still use logic lists as a model. Rust's `BTreeSet` is a set implemented using a b-tree. All elements are unique and arranged in ascending order. Thus, our list model of the b-tree is simply a sorted list in ascending order, where uniqueness of elements is preserved. The abstraction function α that converts the `BTreeSet` to our list model is simply an in-order traversal.

The first example to be illustrated is again the specification of the insertion operation with signature:

```
1 pub fn insert(&mut self, value: T) {...} Rust
```

The signature of the abstract insert operation on our model lists is the same as for `LinkedList::insert`. The specification of `abs-insert` for `BTreeSet`, however, differs from that of `LinkedList`, as we must maintain ordering and uniqueness of elements:

$$\{xs_0. xs_0 = \text{dedup} (\text{sort } xs_0 <) \} \text{abs-insert } \{xs_0 \ x \ xs. xs = \text{model-insert } xs_0 \ x\} \quad (2.6)$$

Again, x is the element to be inserted, and xs_0 and xs are lists modelling the container (via the in-order traversal function α) before and after the `abs-insert` operation respectively. We place an assertion $xs_0 = \text{dedup} (\text{sort } xs_0 <)$ in the precondition requiring that the model xs_0 to be a sorted list of unique elements. While this precondition should always be satisfied by an in-order traversal of a valid b-tree, we do not want our abstraction to constrain the implementation's behaviour if the data invariants of the b-tree are violated — given a malformed b-tree, the implementation should be free to return any result. Because the semantics of `abs-insert` are the maximal relation satisfying this specification, this abstract operation contains all possible behaviours of the concrete implementation if this precondition is violated. The `model-insert` here is simply an insertion operation defined on a sorted list of unique elements:

```
1 (define (model-insert xs x) (dedup (sort (append xs (list x)) <))) Rosette
```

We can also provide specifications for abstract operations that observe the ordering of elements in a `BTreeSet`, such as those operations from the `IndexableT` trait, since there is a one-to-one correspondence between each element's position in a `BTreeSet` and its position in the model list abstracted from the `BTreeSet`. For instance, we provide the specification of the operation `BTreeSet::first`, which is the operation obtaining the first (and also the minimal) element of a `BTreeSet` with signature:

```
1 fn first(&self) -> Option<T> {...} Rust
```

We again provide the signature of its corresponding abstract operation:

```
1 abs-first: List<T> -> (List<T>, T?)
```

Like `LinkedList::contains` in listing 2.4, this type includes a returned list, as Primrose does not consider the immutability of `&self` in the Rust type signature above. We again include the requirement that the container is unchanged in the specification:

$$\{xs_0. xs_0 = \text{dedup} (\text{sort } xs_0 <) \} \text{abs-first } \{xs_0 \ xs \ x. (xs, x) = \text{model-first } xs_0\} \quad (2.7)$$

Here, `model-first` is defined as a function that returns the first element of the list, is present, along with the list itself:

```

1 (define (model-first xs)                                Rosette
2   (cond
3     [(null? xs) (cons xs null)]
4     [else (cons xs (first xs))]))

```

As before, our precondition includes the assumption that the model xs_0 abstracted from the `BTreeSet` contains unique elements that are sorted in ascending order.

2.5.4 The Library Specification of A HashSet

A tree implementation of a set maintains its elements in a fixed ascending order, and the ordering of our abstract list model simply reflects the ordering of the elements in the tree. However, some container implementations do not have a fixed ordering of elements. For instance, the `HashSet` in Rust is a set implementation using a hash algorithm which is randomly seeded. Despite the implementation storing elements in an unspecified order, we may still safely use a sorted, ascending list of unique elements as our abstract model of a `HashSet`: Our abstraction function α merely collects all elements from the `HashSet` into a list and then sorts them into ascending order.

Since the ordering of elements in our list is now different from the ordering of elements in the `HashSet`, the developer may specify properties relating to the ordering of elements, such as `ascending`, that are not satisfied by the implementation, but are trivially satisfied by the abstraction function. This would lead to `HashSet` being considered a valid choice for an `ascending` container. However, Primrose prevents this by the checking of syntactic properties. The `HashSet` type does not implement any trait with operations that allow the ordering of its elements to be observed.

Therefore, in applications for which the ordering of elements is important, `HashSet` is never a valid choice. The selection process of valid implementations according to traits is discussed in section 2.6.1.

If a library developer decides to write a `HashSet` with operations that leak ordering, they can provide a nondeterministic library specification for such a `HashSet` that can still be used by Primrose in the selection process.

For the operations defined on `HashSet` and `BTreeSet`, such as `insert`, `remove` and `contains`, the specifications of both implementations are identical—after all, the only observable difference between the implementations is performance—but the specification for `HashSet` lacks operations that observe the ordering of its elements, such as `first` or `last`.

2.5.5 Abstracting Over Implementation Details with Library Specifications

Since the basic container operations of both `HashSet` and `BTreeSet` have the same externally observable behaviour, we can use the same specifications for both implementations. There are many such cases where specifications can be re-used: For instance, we provide two implementations of an ascending vector: `SortedVec` and `LazySortedVec`. `SortedVec` maintains the ascending order of elements inside the vector on insertion (*eager*) and `LazySortedVec` instead sorts elements whenever the vector is accessed (*lazy*). Since both implementations share the same externally observable behaviour, we use the same model for both implementations: A list with elements sorted in ascending order. Also, their operations are specified with the same set of model operations. For the eager implementation, the abstraction function α simply collects all its elements into a list. For the lazy implementation, in addition to collecting all elements into a list, the abstraction function α also sorts elements into ascending order.

2.6 Selecting and Ranking Implementations

Before ranking container implementations by performance or other non-functional metrics, Primrose must first identify all implementations that comply with the property specifications provided by the application developer.

2.6.1 Selecting Container Implementations Satisfying Syntactic Properties

The first step of selecting valid implementations is to select concrete container implementations from the library that satisfy required syntactic properties in a property specification, which is straightforward. Primrose simply picks concrete container implementations that implement the traits required by the property specifications.

Listing 2.6: A property specification composing semantic and syntactic properties: `ascending`, `ContainerT`, and `IndexableT`

```

1 property ascending { \c -> (for-all-consecutive-pairs c leq?) }   Primrose
2 type AscendingIndexableCon<T>
3   = { c <: (ContainerT, IndexableT) | (ascending c) }
```

For instance, shown in listing 2.6, suppose that in a property specification, an application developer requires a container type implementing traits `ContainerT` and `IndexableT`, the elements of which are sorted in ascending order.

In Rust’s collections library, there are four concrete container implementations `Vec`, `LinkedList`, `BTreeSet`, and `HashSet`, where `Vec`, `LinkedList`, and `BTreeSet` implement both required traits while `HashSet` does not implement the trait `IndexableT`. Clearly, `HashSet` does not satisfy all required syntactic properties. Therefore, `HashSet` is ruled out as a possible implementation for `AscendingIndexableCon<T>`. The implementation for `AscendingIndexableCon<T>` is then selected from the remaining `Vec`, `LinkedList` and `BTreeSet` types by checking if the library specifications satisfy the required semantic property, `ascending`.

2.6.2 Selecting Container Implementations Satisfying Semantic Properties

After gathering container implementations with required syntactic properties, Primrose selects the ones that satisfy the required semantic properties from these candidates. As discussed in section 2.5, our library specifications abstract over the concrete container implementations, describing their externally observable semantics in a compact and tractable format. Primrose performs this selection process by encoding the property specifications as verification conditions against the candidates’ library specifications in Rosette, to be discharged by an SMT solver in Rosette’s backend.

To generate the required verification conditions, Primrose first translates the required semantic properties, given in the specification language of Primrose, into definitions in Rosette that can be used by the solver. The container type `Con<T>` is resolved into the model type used in our library specifications, i.e., a logic list. For instance, the generated code according to the property `ascending` from listing 2.6 is:

```
1 (define ascending (lambda (c) (for-all-consecutive-pairs c leq?)))Rosette
```

With these Rosette definitions, Primrose generates verification conditions. For example, to check if `BTreeSet` is `ascending`, Primrose checks that the semantic property `ascending` is an invariant held across each operation defined for `BTreeSet`. For instance, for the insertion operation, specified by listing 2.6 in section 2.5.3, it checks that the property `ascending` is preserved by any execution that satisfies its precondition and its postcondition:

Recall that xs_0 and xs are model lists abstracted from the `BTreeSet`, specifically, xs_0

$$\forall x_{s_0} \ xs \ x. \frac{xs_0 = \text{dedup}(\text{sort } xs_0 <) \quad xs = \text{model-insert } xs_0 \ x}{\text{ascending } xs_0 \Rightarrow \text{ascending } xs}$$

(where: $\exists x_{s_0}. \text{ascending } xs_0 \wedge xs_0 = \text{dedup}(\text{sort } xs_0 <)$)

Figure 2.5: The rule for checking the operation `BTreeSet::insert` against ascending

is the model for the input `BTreeSet`, and `xs` is the model for the resulting `BTreeSet` of `BTreeSet::insert`. The model operation `model-insert` specifies the behaviour of `BTreeSet::insert`'s corresponding abstract operation. Given the rule shown in figure 2.5, the solver attempts to find a counterexample, i.e., for all input models `xs0` that satisfy the semantic property `ascending`, the solver tries to find a resulting model of the operation that does not satisfy the property. If there is no such counterexample found, the solver will conclude that the operation `BTreeSet::insert` satisfies the property `ascending`.

This search for a counterexample is parameterised by a *model size*, which denotes the maximum size of the input list `xs0` considered by the solver. This parameter is configurable by the application developer using Primrose, and its impact on Primrose's selection time is evaluated in section 2.7.2.

The rule contains a side condition stating that there should be no contradiction between the required semantic property and the precondition of the operation. This side condition is important for ensuring that the solver does not search for a counterexample in an empty search space then falsely conclude that the absence of the counterexample means that the property holds. The side condition requires that there exists at least one model that satisfies both the precondition of the operation and the required semantic property. Without the side condition, the rule is unsound.

In general, the library specification of each operation takes the form:

$$\{\phi(xs_0, \vec{u})\} \text{ op } \{\psi(xs_0, xs, \vec{v})\} \quad (2.8)$$

where `xs0` is the (abstract list model of the) input container and `xs` is the result of the operation `op`. The sets of variables \vec{u} and \vec{v} denote any additional variables involved in the specification, such as additional inputs or outputs to the operation. The general form of the verification condition Primrose generates for the SMT solver, to check if an operation `op` satisfies a property *P*, is given in figure 2.6.

For our `BTreeSet` example, Primrose checks these verification conditions for each operation of `ContainerT` and `IndexableT`—the traits implemented by `BTreeSet`. Since the

$$\forall x s_0 \ x s \ \vec{u} \ \vec{v}. \frac{\phi(x s_0, \vec{u}) \quad \psi(x s, \vec{v})}{P(x s_0) \Rightarrow P(x s)} \quad (\text{where: } \exists x s_0 \ \vec{u}. P(x s_0) \wedge \phi(x s_0, \vec{u}))$$

Figure 2.6: The rule for checking an operation against a property

property `ascending` is satisfied by all operations, Primrose concludes that the `BTreeSet` is a valid implementation for the required container type `AscendingIndexableCon<T>`.

The same checks are also run for the other two candidates that satisfy the required syntactic properties (`Vec` and `LinkedList`) but they do not satisfy the required semantic property `ascending`. Therefore, Primrose concludes that only `BTreeSet` is a valid implementation for the required container type `AscendingIndexableCon<T>`.

2.6.3 Handling Interactions Between Semantic and Syntactic Properties

In this section, we discuss how Primrose selects library implementations with semantic and syntactic properties, such as the stack container `StackCon<T>` from section 2.4.3, where the operations `push` and `pop` specified in the trait `StackT` (listing 2.2) are made available to the semantic property `lifo` (listing 2.3).

Firstly, Primrose generates the definition of semantic property `lifo` in Rosette, where the operations `push` and `pop` are now replaced with their model operations:

```
1 (define lifo (lambda (c) (forall (list x)
2   (equal? (cdr (model-pop (model-push c x))) x))))
```

The specific model operations `model-pop` and `model-push` are supplied to this definition for each candidate type considered by Primrose. Recall that our library specifications state that these model operations exactly specify the intended behaviour of every library operation, which means that these model operations can be used here to express assertions about the interaction between operations such as `push` and `pop`. Such assertions will, by virtue of forward simulation, also apply to the concrete implementations of the data type.

To illustrate the selection process, suppose a library developer provides two implementations that implement `push` and `pop`. The first one is a last-in-first-out imple-

mentation, where the library specification of push and pop is:

$$\{xs_0. \text{true}\} \text{abs-push}_1 \{xs_0 \ x \ xs. \ xs = \text{model-push } xs_0 \ x\} \quad (2.9)$$

$$\{xs_0. \text{true}\} \text{abs-pop}_1 \{xs_0 \ xs \ x. \ (xs, x) = \text{model-pop } xs_0\} \quad (2.10)$$

And the model operations are defined as:

```
1 (define (model-push-front xs x) (append xs (list x)))           Rosette
2 (define (model-pop xs)
3   (cond [(null? xs) (cons xs null)]
4         [else (cons (take xs (- (length xs) 1)) (last xs))]))
```

With these two model operations, the solver can verify that this library specification satisfies the semantic property `lifo`.

By contrast, the second implementation is a first-in-first-out implementation. The library specification of push and pop appears similar:

$$\{xs_0. \text{true}\} \text{abs-push}_2 \{xs_0 \ x \ xs. \ xs = \text{model-push } xs_0 \ x\} \quad (2.11)$$

$$\{xs_0. \text{true}\} \text{abs-pop}_2 \{xs_0 \ xs \ x. \ (xs, x) = \text{model-pop } xs_0\} \quad (2.12)$$

However, the model operations have different semantics:

```
1 (define (model-push-end xs x) (append (list x) xs))           Rosette
2 (define (model-pop xs)
3   (cond [(null? xs) (cons xs null)]
4         [else (cons (take xs (- (length xs) 1)) (last xs))]))
```

With these two model operations, the solver correctly concludes that this library specification does not satisfy the semantic property `lifo`, and Primrose does not consider this implementation as a valid choice for the container `StackCon<T>`.

2.6.4 Code Generation and Ranking Implementations by Performance

Once Primrose has selected the valid container implementations, it will generate a Rust program for each valid candidate by resolving the property specification into the selected container implementation. In figure 2.3, we show a simplified version of generated programs where property specifications are directly replaced with concrete implementations, in practice Primrose carefully generates Rust's trait objects to encapsulate the concrete implementation and exposing only those operations in Rust traits which are specified as syntactic properties.

As a proof-of-concept implementation, the current Primrose prototype ranks the generated Rust code for each valid implementation by executing all candidates and measuring their runtime on some test input data. We anticipate adopting more sophisticated ranking techniques, such as the ones discussed in the related work, in the future. Our existing prototype of Primrose focuses on enabling application developers to specify their functional requirements, and automating the selection of valid container implementations.

2.7 Evaluation

For Primrose to be feasibly used as a programming tool, it must be practical to ensure that the container implementations indeed satisfy their library specifications, and the selection process itself must not take a prohibitively long time. The evaluation demonstrates feasibility in both of these aspects. All measurements are conducted on a MacBook Pro with 32 GB of RAM and a 2.4 GHz 8-Core Intel Core i9 processor.

2.7.1 Correctness of Container Implementations w.r.t Their Library Specifications

To ensure the selected implementations are correct, we validate our Rust container library implementations against the library specifications using property-based testing (Claessen and Hughes, 2000). We use the framework `proptest` (AltSysRq, 2022) for encoding and performing the tests.

Firstly, we encode the model list with its operations in Rust. Specifically, we encode the model list from Rosette as an immutable `ConsList` (Stokke, 2022) in Rust, along with all its operations. Then we implement the abstraction function α for each container implementation. Like Chen et al. (2017, 2022), we encode the forward simulation obligation for the library specification of each operation as assertions in a test.

For each test, 100 test inputs are randomly generated. For our library with eight container implementations, in total 7200 inputs are tested in 7.315 seconds. We conclude that with the existing testing framework, we are able to validate the functional correctness of our container implementations w.r.t. our library specifications efficiently, ensuring that implementations selected by Primrose are correct.

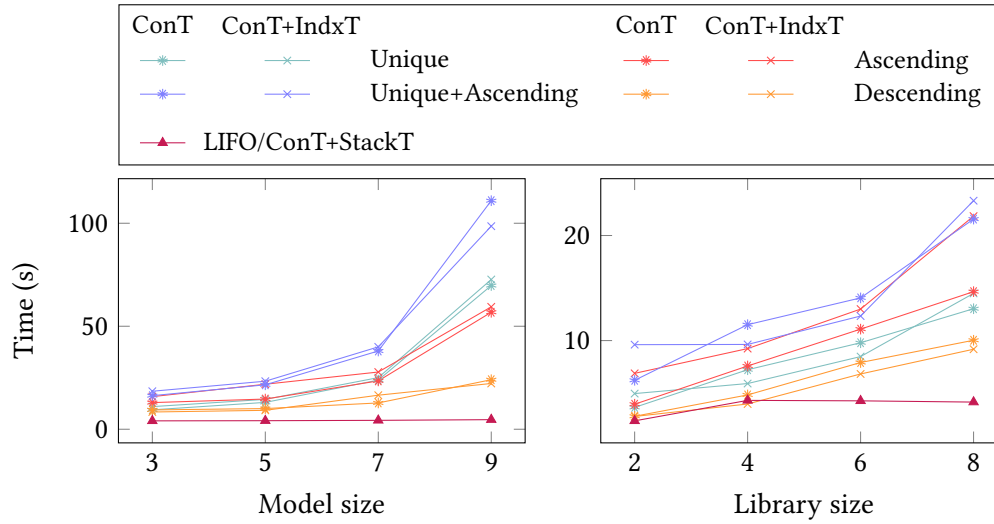


Figure 2.7: Primrose’s efficiency of selecting implementations for different properties

2.7.2 Evaluation of Primrose’s Selection Time

For Primrose to be practical, it must perform selection with a reasonable time, even though, as a pre-processing tool, it does not have to be invoked on every compilation run. After the initial invocation, it will only be invoked if the property specification or any library specifications are changed.

The efficiency of the SMT-based selection time is mainly determined by two factors: the *model size* and the *library size*, which together define the search space in which the solver attempts to find a counterexample. If a counterexample is found, Primrose will conclude that the library specification does not satisfy the required semantic property. We expect the solver time to grow linearly with the number of container implementations from which we select (library size) and non-linearly with the model size, which is the length of the input model list to the abstract operation, as this should grow the search space polynomially.

Figure 2.7 shows the measurements of Primrose’s selection time. The left side shows that the selection time, for a fixed library size of eight implementations, increases with the model size. The right side shows that for a fixed model size of five, the selection time increases linearly when the library size is increased.

The complexity of the property specifications and the number of satisfying implementations are also factors that affect the efficiency of the selection, since they determine how difficult it is for the solver to find a counterexample. For example, since the definition of `lifo` has constant complexity, the model size and library size do not affect its selection time as much as for properties with high polynomial complexity

such as `unique` and `ascending`. None of our example containers satisfy the property `descending`. As SMT solvers are faster at finding a counterexample than exhaustively proving that no counterexample exists, the selection time for `descending` is faster than for `ascending`, despite both properties having the same algorithmic complexity.

The selection process is always completed within 30 seconds with a model size of three and the full library of eight implementations. Although an increase in model size raises selection time quickly, in practice, a model with size of more than five is not required to admit counterexamples for most conceivable semantic properties that the application developer may write. This is based on the small scope hypothesis in Alloy Jackson (2012). We conclude that Primrose is feasible for medium-size libraries.

2.8 Discussion of Limitations

Primrose’s prototype implementation has some limitations that we discuss here. Primrose currently covers properties of sequential containers like lists and sets, and we have not yet looked into associative containers like maps and dictionaries. However, we believe it should be possible to characterise them with the same technique: application developers using syntactic properties to describe desired operations and semantic properties to state predicates that should be held by keys and values, and library developers providing library specifications using a list model with key-value pairs as elements.

The other limitation of our current implementation is that we implicitly require all elements inside a container to have some ordering for them to be comparable using `leq` and `geq`. In the future, we should allow application developers to state if the elements inside the container are comparable or have ordering by enriching the syntax and type system of our property specification language.

2.9 Related Work

Refinement Types Refinement types, first introduced for ML (Freeman and Pfenning, 1991), are types enriched with logical predicates, often from an SMT-decidable logic (Bierman et al., 2010), allowing programmers to express rich logical constraints in the type system and automatically check them. Refinement types have recently been implemented in languages such as Haskell (Vazou et al., 2013, 2014) and F* (Swamy et al., 2016), supporting very rich specifications suitable for verifying the correctness

of programs. While the syntax of Liquid Haskell inspires our design of the syntax of property specifications, we use refinement types not for verification, but for data abstraction, allowing application developers to specify their semantic requirements for the selection process.

Abstract Data Types and Formal Methods An abstract data type is characterised by the operations in its interface, rather than the details of its implementation (Liskov and Zilles, 1974). This data abstraction allows a separation of concerns, freeing application developers from having to consider the internal implementation details of a data type, instead allowing them to consider only the externally observable semantics of each operation. Our work advances this area of data abstraction, allowing abstraction to be maintained even when creating an instance of the data type, so that programmers can work on the level only of requirements, without needing to consider implementation details.

Existing work in algebraic specifications (Guttag, 1976; Wirsing, 1990) provide a formal definition of abstract data types where the semantics of operations are specified with a set of equational axioms. By contrast, our library specifications are model-based. As mentioned in section 2.5.1, this allows us to easily ensure completeness of library specifications. There exist many formal modelling tools that facilitate model-based specification of abstract data types and software systems more generally, for example Z (Spivey, 1989), VDM (Jones, 1990), and most recently Alloy (Jackson, 2006). While these tools allow application developers to formally analyse and explore the software design space, including formal reasoning about abstract data types, they work purely on the level of models and do not typically connect to actual code, as Primrose does.

Performance-Oriented Selection Techniques Many techniques for design space exploration, particularly machine learning techniques, have been applied in compilers (Fursin et al., 2011) to selected performance optimization techniques (Cavazos et al., 2007) using various characteristics as features that are then used to rank the performance of multiple implementations (Siegmund et al., 2012). Many dynamic selection techniques have been developed for assisting the selection of performant containers, based on different evaluation criteria such as workload data (Costa and Andrzejak, 2018), architectural concerns (Jung et al., 2011) and runtime metrics (Shacham et al., 2009). In addition to dynamic container selection, CoCo (Xu, 2013) is tool al-

lowing safe online switching.

None of these techniques, however, provide a general scheme to allow application developers to specify desired behaviour, instead, they purely focus on selecting between multiple, pre-known, valid container implementations. Such techniques could be incorporated into Primrose’s ranking process, and are highly complementary with our work.

2.10 Conclusion

This study presents Primrose, a language-agnostic tool for selecting the best performing container implementation that satisfies a set of desired properties. Semantic properties provide application developers with a powerful, novel abstraction to describe the expected behaviour of a container as a refinement of the container data type. Semantic properties nicely complement syntactic properties (i.e., traits), allowing developers to specify the programming interface and behaviour of a container without committing to a concrete implementation. Primrose automatically selects the set of valid container implementations for which the *library specifications*, written by the developers of container libraries, satisfies the specified properties. Finally, Primrose ranks the valid library implementations based on their runtime performance.

To summarise, this study makes the following contributions:

- We present Primrose, a language-agnostic tool for selecting valid container data type implementations based on *properties* used to describe their behaviour and programming interface, and ranking them based on their performance.
- We show a new application of refinement types not—as previous work did—for verification purposes, but to raise the level of abstraction for application developers and to improve the runtime performance of applications with container data types.
- We present formal specifications of the Rust container data types from the standard library and customised implementations and used them to valid the implementations by property-based testings. These specifications could also be used in the future to formally verify the correctness of their Rust implementations.

We have applied techniques from verification and formal methods in a new way, raising the level of abstraction by freeing developers from the burden of choosing con-

crete container implementations. Instead, application developers can specify their expected behaviour using semantic properties—a highly general abstraction technique. We provide a methodology to specify container libraries with library specifications, and describe our mechanism to check semantic properties against these specifications using SMT solvers. We implement Primrose for Rust and specify eight Rust container implementations. We show that Primrose is a practical tool that can be feasibly integrated into a programmer’s workflow.

Primrose is already effective at selecting data types based on *functional requirements*, but significant future work remains in selecting and ranking based on *non-functional* metrics. We intend to significantly improve the ranking method used by Primrose by using more sophisticated techniques, including machine-learning-based solutions and techniques that optimise for multiple metrics, such as run time and memory use. Integrating Primrose with a dynamic technique that changes container implementations on-the-fly based on runtime performance measurements would also be interesting.

Epilogue

Addressing the issue that application developers are forced to *over-specify* what they mean in a program by committing to concrete implementations, the solution proposed by this chapter, which is also the core theme of this chapter, is to design declarative *specifications*. Library specifications specify the functional correctness of implementations, while property specifications including both syntactic specifications and semantic specifications form an abstraction over concrete implementations, allowing application programmers to better express what they want a computer to do.



In the end, I would like to enclose this chapter with some high-level observations and reflections. There is often a trade-off between concise abstractions and rich expressiveness. The concrete implementations of containers convey both functional and non-functional requirements. Abstracting over these implementations, we have seen that there are concise and accurate ways to express functional requirements as specifications, which can be used to validate the container implementations. However, unlike implementations, these specifications can not able be used to express the non-functional requirements since it is not an easy task to accurately specify these non-functional requirements, such as algorithmic time and space complexities. In terms of expressiveness, we have observed that there is often a trade-off: Abstractions have the advantage of better representing the characteristics of functional properties

in a concise way. However, because of their nature of omitting details of implementations, abstractions are not expressive enough to characterise non-functional properties like runtime performance. Therefore, other compile-time and runtime optimisations techniques other than purely formal specifications are necessary for guiding the performance-orientated selection process.

Chapter 3



Oxidising Remote Procedure Calls

A Universal Method Invocation Library for Rust




The building is circular. The apartments of the prisoners occupy the circumference. You may call them, if you please, the cells. The apartment of the inspector occupies the centre, you may call it if you please the inspector's lodge. To cut off from each prisoner the view of every other, the partitions are carried on a few feet beyond the grating into the intermediate area, such projecting parts I call the protracted partitions. These windows of the inspector's lodge open into the intermediate area, in the form of doors, in as many places as shall be deemed necessary to admit of his communicating readily with any of the cells.

— Jeremy Bentham “Panopticon or the Inspection-House”



Prologue

 ECALL that in chapter 1, we have briefly discussed the conceptual question that we would like to ask: How to intuitively understand *distributed programs* using the same conceptual model as *monolithic programs*? It would be beneficial to view a monolithic programs as an abstraction of distributed program, specifying the intended behaviours of the invocations and resource usage in the distributed program while abstracting away message-passing details, since it

would make the process of migrating monolithic programs into a distributed setting straightforward and simplify the process of implementing a distributed system.

In this chapter, we discuss in detail our design, implementation and formalisation of a *universal method invocation* (UMI) library in Rust, which supports location transparency, allowing a monolithic program to be migrated into a distributed design with minimal syntactic modification and preserves the semantics of the original program.

As a metaphorical illustration, the core idea of our attempt for conceptually modelling and understanding distributed programs as monolithic programs resembles a *panopticon*, where the distributed resource management is governed by a monolithic design of Rust’s ownership and borrow checking system.

3.1 Introduction

Distributed computing has extensive application in areas such as cloud computing, big data processing, web services, and blockchain systems, driving the development of modern, large-scale, and resilient software systems. Distributed systems offer many significant advantages such as scalability, fault tolerance, resource sharing, and geographical distribution.

However, compared to monolithic systems, distributed systems are more complex and challenging to design, implement, test and debug due to the necessity for coordination, synchronisation, and communication among distributed components. Therefore, it is common practice to start with a monolithic design when implementing a system, as this approach simplifies the initial implementation and deployment process. Such a monolithic system needs to later be re-structured and migrated into a distributed design when it needs to be expanded into a larger scale. However, it still requires non-trivial effort to be put into the migration of a monolithic system into a distributed design.

3.1.1 Contributions

To address these issues in the design and implementation of a distributed system as well as migrating a monolithic systems into a distributed setting, we propose our design of a UMI library in Rust. The design of the UMI library shares the same underlying idea of the *remote procedural call* (RPC) (Nelson, 1981), where a method invocation on an object can be executed on a different node within the same network, abstracting

over the underlying message-passing details. Such a design allows programmers to model a distributed system focusing on *what* functional features are required instead of *how* these functional features are achieved via complicated network communications. Moreover, it allows applications to be migrated from a monolithic design to a distributed architecture without massive changes to source code or the needs of high-level expertise in microservices. In addition, by choosing Rust as the language for implementing the UMI framework, we are able to avoid distributed memory management hassles like distributed garbage collection while extending Rust’s memory safety and data-racing free guarantees into the distributed setting.

In summary, we make following contributions:

- We provide a usable *Rust implementation of the UMI framework* (section 3.3).
- We formalise the *structural operational semantics* based on Pearce (2021)’s featherweight Rust (FR) for a core calculus of monolithic Rust programs and distributed Rust programs written in the UMI framework (section 3.4).
- We prove a *location transparency theorem*: With the UMI framework, when a monolithic program is deployed to multiple nodes, its semantics is preserved (section 3.4.3).

3.1.2 Limitations

There are some limitations of this project worth mentioning upfront. Firstly, the UMI framework does not handle network communication errors. Since technically it is difficult to handle these errors in a distributed program while maintaining the same interface of its monolithic counterpart without the language feature throwing and handling exceptions. Also, as we plan to integrate this framework within micro-services platforms, where server errors are managed by cloud service providers. Supervision strategies can be employed to take snapshots and restart from failures, ensuring these issues do not pose a critical problem for the UMI framework’s design.

In addition, both the implementation and the formalisation of the UMI framework are deterministic and sequential. In the current stage, such a design decision is sufficient to demonstrate of core concepts of the UMI framework including location transparency and memory safety. However, as potential future work, it would be nice to model the UMI framework that accounts for concurrency.

Moreover, there are limitations in the formal system we have presented, for instance, functions and structs are missing. These issues are caused by the formal system we build upon. Many different styles of formal systems of Rust have been surveyed for this project however it is rather hard to find a formal system that models sufficient core features of the surface language of Rust. We will discuss related formalisation of Rust in section 3.5. In order to have a better formalisation of our system, it is required to have a better formalisation of the surface language of Rust, which is out of the scope of this project.

Before stepping into the detailed discussion of the UMI library, in the next section, we will give a high-level overview of remote procedure calls as well as Rust.

3.2 Background

In distributed computing, a RPC allows a method invocation to be executed on another computer on a shared network. One application of RPC is that in an object-oriented programming paradigm, it enables a method to be invoked on an object stored on a different machine and exchange data across the network. Such a remote method invocation has the same coding as a local invocation, without the programmer explicitly coding the details for the remote interaction. However, it is hard to support *location transparency*, i.e., in most existing frameworks (e.g., Java RMI), remote invocations do not have *the same semantics* as local invocations. In addition, *memory management* is hard in a distributed setting, for example, distributed garbage collection is complicated.

Rust (Klabnik and Nichols, 2018) is a high-level system programming language which *guarantees memory safety* and *prevents data races* by its *ownership system* for memory management and *borrow checker* for tracking object lifetime of all references in a program during compilation. Since Rust’s semantics guarantees memory safety, we can extend such guarantees to the distributed computing setting, allowing us to design a RPC framework that provides safe remote method invocations.

3.2.1 Remote Procedure Calls

A RPC allows a computer program to request a service from another program located in a different address space, which could be on the same machine or on a different machine across a network. It is a form of client-server communication, where the

requesting program is the client, and the service-providing program is the server.

The basic idea behind a RPC system is to make a remote invocation appear like a local invocation, abstracting away the underlying communication mechanisms like message passing and network protocols and simplifying distributed computing by providing a familiar programming model. When a client program calls a procedure, a RPC system will handle the task of transferring the procedure call request to the remote server, along with any necessary parameters or data. The server then executes the requested procedure and sends the results back to the client.

RPCs are particularly useful in distributed systems, where different components of an application are running on separate processes or machines. It allows these components to communicate and share resources efficiently, as if they were part of a single program. There are many common applications of RPCs. For instance, RPCs are used in distributed file systems, such as Network File System (NFS), to enable clients to access and manipulate files on remote servers transparently. In remote database access, RPCs facilitate remote access to databases, allowing client applications to execute queries and retrieve data from remote database servers. In designing web services, RPCs form the basis of many web service protocols, such as SOAP (Simple Object Access Protocol), which allows applications to communicate over the internet using XML-based messaging. In modern microservices architectures, RPCs are often used for inter-process communication between different microservices, enabling them to collaborate and share functionality. In object-oriented programming, RPCs are commonly implemented as remote method invocations (RMI), enabling objects on different machines to interact with each other. The core feature of RMIs is that objects can interact with each other by invoking methods and passing data across the network. This is the application domain that we focus on in this study.

3.2.2 Rust

As a system programming language with emphasises on safety, performance, and concurrency, Rust is designed to prevent some common programming errors, such as data races and dereferencing null pointers. Rust achieves these goals through its distinctive features of the ownership system, borrow checking, and lifetimes.

In Rust, each value has a variable designated as its *owner*. Each value can only have one owner at a time, and when the owner goes out of scope, the value is dropped, i.e., deallocated from memory. This ownership model ensures that resources are managed

correctly without the need for a garbage collector. The ownership system is fundamental to Rust and serves as the basis for its memory safety.

Rust allows functions and data structures to create references to values *without* taking ownership. This is called *borrowing*. When a value is borrowed, the original owner cannot modify the value until the borrowing ends. The *borrow checker* is part of Rust's compiler, which ensures that references are used safely and do not result in dangling pointers or other memory issues. Borrowing can be either mutable or immutable, where mutable references have additional constraints to prevent data races and undefined behaviour.

Lifetimes in Rust express how long references should be valid. They assist the borrow checker in ensuring that references do not outlive the data they point to. Rust uses lifetimes to prevent dangling references, which is important for memory safety. Lifetimes are explicitly annotated or inferred. They work alongside the ownership system and borrow checking to maintain memory safety and prevent data races.

With the design of the ownership system, borrow checking mechanism, and lifetimes, Rust enforces strict memory safety guarantees, i.e., that all references point to valid memory, without requiring a garbage collector. These features also ensure that Rust programs do not have data races by allowing only one mutable reference at a time or multiple immutable references.

3.3 The Implementation of the Rust UMI Library

In the section, we present our implementation of the UMI framework as a library in Rust. With such a library, a monolithic program can be migrated into a distributed program while preserving the semantics of the original monolithic program.

3.3.1 Overview

To give a high-level overview of the design, in figure 3.1, we introduce an example of migrating a monolithic program into a distributed setting, by adding the *macros* provided by the UMI library. In this example program which allocates instances of the struct `A` and calls methods on them, the macro `#[proxy_me]` implicitly translates the declared type `A` from a struct that can only refer to local resources to an enum that can either hold local resources or be a *proxy* that refers to resources held on a remote node. The initialisation method is translated by the macro `#[umi_init]` to create an

<pre> 1 #[proxy_me] 2 struct A { arg: u32 } 3 impl A { 4 #[umi_init] 5 new(arg: u32) -> A { A {arg: arg} } 6 #[umi_struct_method] 7 by_value(&self, a: A) {...} 8 #[umi_struct_method] 9 by_ref(&self, &a: A) {...} 10 #[umi_struct_method] 11 by_mut_ref(&self, &mut a: A) {...} 12 ... 13 } </pre>	<pre> 1 fn main() { 2 let a_remote = 3 remote!(addr, A::new(10)); 4 5 let a_local1 = A::new(1); 6 let a_local2 = A::new(2); 7 let mut a_local3 = A::new(3); 8 9 a_remote.by_value(a_local1); 10 a_remote.by_ref(&a_local2); 11 a_remote 12 .by_mut_ref(&mut a_local3); 13 } </pre>
--	---

Figure 3.1: Migrating A Monolithic Application into A Distributed Setting with UMI

instance of the enum `A` instead of an instance of the struct `A`. Other methods are also translated by macros to allow both an invocation on a local instance of `A` and an invocation on a proxy of `A`. To create a proxy instance, the macro `remote!(address, ...)` is used, while the syntax of the initialisation of a local instance is unchanged. The invocations of the methods defined for translated struct `A` take the same form of the invocation of those original methods.

An invocation on a proxy is encapsulated into a serialised message and sent to the destination node of which the address is the address stored in the proxy, and then the message is deserialised and the invocation is executed at the destination. After the execution, the result of the invocation is again put into a serialised message and passed back to the calling node to be deserialised.

3.3.2 The Design of the Translation

As we have seen in the example discussed above, the syntax of a monolithic program is translated into a distributed program by a set of macros. For a declared struct, the macro `#[proxy_me]` performs the translation:

$$\text{struct } A \{ \text{fields} \} \rightsquigarrow \text{enum } A \{ \text{Local} (\text{fields}), \text{Remote} (\text{Address}, \text{ID}, \text{IsOwner}) \}$$

where the *Address* is the address of the node which stores the resource of a proxy, the *ID* is the identifier of a proxy's resource in the resource table that will be discussed in section 3.3.3, and *IsOwner* denotes whether a proxy is an owned reference or a

borrow reference. This macro can translates an enum to allow it to represent a proxy by adding a new constructor which is the proxy:

$$\text{enum } A \{ \text{variants} \} \rightsquigarrow \text{enum } A \{ \text{variants}, \text{Remote} (\text{Address}, \text{ID}, \text{IsOwner}) \}$$

The translation of a enum does not affect its initialisation method, however, the translation of a struct requires its initialisation method to be changed accordingly — instead of creating an instance of a type which is a struct, an instance of a *Local* variant of an enum is created. For instance, in the example shown in figure 3.1, the `new(arg:u32)` method is translated by `#[umi_init]` into:

```
1 new(arg: u32) -> A { A::Local {arg: arg} }
```

The macro `#[umi_struct_method]` performs the translation of other methods of a struct. For instance, the method `foo1(&self, a:A)` is translated into:

```
1 fn foo1(&self, a: A) {
2     match &self {
3         Local(...) => { /* do something */ },
4         Remote(...) => { /* remote do something */ }
5     }}
```

Note that within the pattern matching block for the *Remote* variant, the invocation is firstly put into a message and serialised. Then the serialised message is passed to the addressed stored in the proxy, and get deserialised and executed. The result is again put into a message and get serialised. Once it is returned back to the original node, the result is extracted from the deserialised message. Such a communication process between nodes via sending and receiving serialisation/deserialisation messages is completely generated by the macro, freeing programmers from dealing with the message passing complexity.

As for a method of a translated enum, the macro `#[umi_enum_method]` adds an additional pattern matching block for the proxy variant to the existing pattern matching.

3.3.3 Resource Management

To be able to use the UMI library for executing programs that access and manipulate memories of different nodes within a network, resources and computations need to be made available to and well-managed by all nodes in the network.

Firstly, a node should be able to store resources owned by different machines and deallocate those resources according to their lifetime. In Rust, if some resources are

ID	Resource	Full Path Name	Type Information
0	...	A::new	u32, A
1	...	A::foo1	(&A, A), ()
...

Figure 3.2: A resource table (L) and a method registration table (R)

owned by a reference on the same node, and the reference has reached the end of its lifetime, these resources will be deallocated from the memory. With such a design, resources that are not owned by any reference on the same node are automatically deallocated. However, in our UMI library, while some resources on a node n_1 are not owned by any reference on the same node, they can be owned by a reference on a different node n_2 . Although these resources do not have a local owner, the deallocation should not happen until the remote owner reaches the end of its lifetime. To achieve this goal, on each UMI server, we design a *resource table* shown in figure 3.2 on the left, which has the same lifetime as the server. We used it to identify and manage local resources involved in remote computations. Note that the ID in an entry of the table is the ID field in a corresponding proxy, which is globally unique. If a variable is created locally, it will be put into the table once it is passed into a remote computation. The entry will not be removed until the remote computation finishes. If a variable is created via a remote call, it will be put into table on creation and will be deallocated when its remote owner decides that it should be dropped.

Secondly, we need to make all nodes aware of all methods that can be invoked on a proxy in order to make computations available on all nodes. To achieve this goal, we register all methods that are available for remote invocations in a *method registration table* shown in figure 3.2 on the right, using the `register!(name, arg_types, return_type)` macro. The method registration table holds the full path name, argument types, and return type of methods. When a serialised invocation message, which takes the form of a plain string, is received by a node, the method to be invoked is deserialised and reconstructed according to the type information recorded in the registration table.

3.3.4 Passing Remote Invocations via Messages

As briefly discussed in section 3.3.1 and section 3.3.3, remote invocations and results of executions are implicitly communicated via serialised and deserialised messages

among nodes. We make use of the Serde (serde-rs, 2023) framework to serialise and deserialise these messages and Rust data structures.

There are different types of messages for passing remote invocations. For instance, a remote invocation sent to an receiving node is represented as an invocation message which taking the form of `Message::Invoke(fname, variables, invoke_op)`, where `fname` is the full path name of the method, each variable is annotated with its ownership information (owned or immutably/mutably borrowed), and `invoke_op` specifies the ownership information of the return value. The result of the execution of a remote invocation is passed back to the calling node via a return message taking the form of `Message::Return(return_var)`, where the `return_var` is also annotated with its ownership information. Another important type of messages is the deallocation message which takes the form of `Message::Drop(id)`, where the `id` corresponds to an entry key in the resource table shown in figure 3.2. Such a message instructs some remotely owned resources to be deallocated.

3.3.5 Extending Borrow Checking into Distributed Settings

To execute a deserialised remote method invocation on the node which receives the invocation, the first step is to gather serialised data as well as the ownership information of each variable involved in the method. In this step, we do not perform any reconstruction of these variables; instead, variables are simply prepared in an appropriate format that can be reconstructed during the execution of the method. Such an format is implemented as `Argument`, which keeps the information of the variables related to borrow checking, and stores the data of the variable. The detailed implementation of this step is shown in listing 3.1.

As shown in listing 3.1, a serialised variable has a label indicating that if it is a piece of data copied or moved from the caller (`OwnedLocal`), a remote reference owned by the caller (`OwnedRemote`), a remote reference immutably borrowed by the caller (`RefRemote`), or a remote reference mutably borrowed by the caller (`MutRefRemote`).

Listing 3.1: Gathering variables from an invocation message

```

1 Message::Invoke(fname, variables, invoke_op) => {
2     let mut arguments: Vec<Argument> = Vec::new();
3     for v in &variables {
4         match v {
5             Variable::OwnedLocal(s) =>
6                 { arguments.push(Argument::Serialised(s.clone())); },

```

```

7      Variable::OwnedRemote(serialise_remote, addr, id) => {
8          if addr == &local_address {
9              let (owned, is_ref) = mvtable.remove(id).unwrap().into_inner();
10             let arg_ref = Argument::Owned(owned);
11             arguments.push(arg_ref);
12         } else {
13             arguments.push(
14                 Argument::Serialised(serialise_remote.to_string()); },
15 Variable::RefRemote(serialise_remote, addr, id) => {
16     if addr == &local_address {
17         let borrow = mvtable.get(id).unwrap().borrow();
18         let ptr: *const (Box<dyn Any + Send + Sync>, bool) = &*borrow;
19         unsafe {
20             let back: &(Box<dyn Any + Send + Sync>, bool)
21                 = ptr.as_ref().unwrap();
22             let arg_ref = Argument::Ref(&back.0, back.1);
23             arguments.push(arg_ref); }
24     } else {
25         arguments.push(
26             Argument::RemoteRef(serialise_remote.to_string())); },
27 Variable::MutRefRemote(serialise_remote, addr, id) => {
28     if addr == &local_address {
29         let mut borrow_mut = mvtable.get(id).unwrap().borrow_mut();
30         let ptr: *mut (Box<dyn Any + Send + Sync>, bool)
31             = &mut *borrow_mut;
32         unsafe {
33             let back: &mut (Box<dyn Any + Send + Sync>, bool)
34                 = ptr.as_mut().unwrap();
35             let arg_ref = Argument::MutRef(&mut back.0, back.1);
36             arguments.push(arg_ref); }
37     } else {
38         arguments.push(
39             Argument::RemoteMutRef(serialise_remote.to_string())); }
40     }}} ...
41 }

```

If a variable is serialised data, which is copied or moved from the caller, it will be kept as serialised, since the deserialisation and reconstruction process will happen during the invocation of the method (line 5 — line 6). If a variable is a proxy which is located at the receiver, then it will be obtained from the resource table shown in figure 3.2. According to its ownership information, if it is moved, then the corresponding

entry will be removed from the resource table (line 7 – line 11). If it is immutably borrowed, then the corresponding entry will be immutably borrowed from the table (line 15 – line 23). If it is mutably borrowed, then the corresponding entry will be mutably borrowed from the table and updated after the execution (line 27 – line 36). If an argument is a proxy that is not located at the caller, as shown in three else-cases, from line 12 to line 14, from line 24 to line 26, and from line 37 to line 39, then the proxy will be passed into the method without any additional modification.

Once the information about all variables is gathered and processed, the invocation will be executed and the result will then be sent back to the caller. The implementation of the execution of this invocation is shown in listing 3.2. The method information, mainly ownership and type information of the arguments, and return value of a method are retrieved from the registration table shown in figure 3.2 on the right. During the execution of the method via `f.call(arguments)` shown in line 6, serialised arguments and boxed argument entries retrieved from the resource table are reconstructed according to the registered type information.

The result of an execution is provided in two formats, serialised data and a boxed data entry. These two formats are used according to the required ownership information of the return value. If the method produces an owned result annotated with `InvokeOp::Owned`, whether the serialised data `res` represents some local resources or a proxy, it will be kept as the serialised form and sent back via a return message (line 9 – line 11).

If the method is an initialisation call sent by the macro `remote!(...)` annotated with `InvokeOp::Init`, the boxed entry data will be inserted into the resource table and a unique id will be generated. In the return message, the address of the receiver, the id, and the ownership status which is `true` are included for the caller to construct a proxy that owns such a data entry on the receiver (line 12 – line 18).

For a return value that is an immutably or mutably borrowed reference, there are two situations. If the borrowed reference is local to the receiver, the reference itself will be inserted into the resource table identified by a generated unique id. Such an id and the address of the receiver will be sent back to the caller for creating an proxy that mirrors this borrowed reference (line 20 – line 25 and line 31 – line 36). However, if a borrowed reference is not local to the receiver, meaning it already mirrors a reference on a different node, then it will not be stored in the resource table, instead, the serialised version of it will be sent back to the caller in a return message (line 26 – line 27 and line 37 – line 38).

Listing 3.2: Executing an invocation and returning the result

```

1 Message::Invoke(fname, variables, invoke_op) => {
2     ...
3     let f: &str = &*fname;
4     match lrtable.get(f) {
5         Some(f) => {
6             let ((res, is_local), b) = f.call(arguments);
7             let res_message: Message;
8             match invoke_op {
9                 InvokeOp::Owned => {
10                     res_message = Message::Return(ReturnVar::Owned(res));
11                 },
12                 InvokeOp::Init => {
13                     let id = (SystemTime::now(), m_id_gen.next());
14                     // b is the resource
15                     mvtable.insert(id.clone(), RefCell::new((b, false)));
16                     res_message =
17                         Message::Return(ReturnVar::OwnedInit(local_address, id, true));
18                 },
19                 InvokeOp::Ref => {
20                     if is_local {
21                         let id = (SystemTime::now(), m_id_gen.next());
22                         // b is a reference
23                         mvtable.insert(id.clone(), RefCell::new((b, true)));
24                         res_message =
25                             Message::Return(ReturnVar::RefMirror(local_address, id));
26                     } else {
27                         res_message = Message::Return(ReturnVar::RefBorrow(res));
28                     }
29                 },
30                 InvokeOp::MutRef => {
31                     if is_local {
32                         let id = (SystemTime::now(), m_id_gen.next());
33                         // b is a reference
34                         mvtable.insert(id.clone(), RefCell::new((b, true)));
35                         res_message =
36                             Message::Return(ReturnVar::MutRefMirror(local_address, id));
37                     } else {
38                         res_message = Message::Return(ReturnVar::MutRefBorrow(res));
39                     }
40                 }
41             }
42             response(stream, res_message);

```

```

42     },
43     None => { /* report unregistered function */ }
44 }, ...

```

3.3.6 Extending Lifetime Management into Distributed Settings

Recall that in section 3.3.3, we have briefly introduced storing and deallocating remotely owned resources on a node. In this section we discuss the design and implementation of a remote deallocation in detail.

Listing 3.3: An example of a remote deallocation

```

1 // on caller
2 fn main() {
3     ...
4     // the data of a_proxy is in the table on the receiver with addr
5     // but it is owned by the caller and will be deallocated
6     // when its owner decides to drop it
7     let a_proxy = remote!(addr, A::new(10));
8     ...
9     a_proxy.foo1(...)
10 } // a_proxy is out of scope, its data on the remote machine is dropped

```

In a monolithic Rust program, when a variable that owns some resources reaches the end of its lifetime, in most cases, out of a program’s scope, the resources it owns will be automatically deallocated. We extend this feature to the distributed setting. As illustrated in listing 3.3, when the given proxy `a_proxy` is initialised, some resources are allocated to the receiver node with the address `addr` (line 7). Although these resources do not have an owner on the same node, they should not be deallocated until its remote owner `a_proxy` reaches the end of its lifetime (line 10).

Listing 3.4: The implementation of a remote deallocation

```

1 impl Drop for #name {
2     fn drop(&mut self) {
3         match self {
4             Self::Remote(addr, id, is_owner) => {
5                 if is_owner.load(Ordering::Relaxed) {
6                     let msg = Message::Drop(*id);
7                     send(*addr, msg).unwrap();
8                 }
9             }
10        }
11    }
12 }

```

For monolithic programs, the deallocation is achieved via the drop method in the destructor trait `Drop`, which in most cases is automatically implemented for Rust types. We extend this drop method to handle the deallocation of remotely owned resources. The implementation is shown in listing 3.4. When a proxy that owns some resources on a node reaches the end of its lifetime, a serialised deallocation message is automatically sent to the node that holds these resources.

As shown below, once a deallocation message is received by the targeted receiver, the entry with the corresponding `id` will be removed from the resource table (i.e., the `mvtable` in the listing).

```
1 Message::Drop(id) => { mvtable.remove(&id); }
```

After presenting the design and implementation of the UMI library, in the next chapter, we discuss the formalisation of core concepts of the UMI library base on formalised operational semantics of a core language of Rust, and sketch a proof of the location transparency theorem stating that using the UMI framework, the translation of a monolithic program into a distributed program preserves the semantics of the original program.

3.4 The Operational Semantics

We first provide a small-step operational semantics of a core language of Rust based on Pearce (2021)’s featherweight Rust (FR), which captures the core features of Rust including copy- and move-semantics, owned and immutably/mutably borrowed references, and lexical lifetimes. We then present dFR, which extends FR to include distributed features of the UMI framework such as remote copy- and move-semantics as well as remote references. We show that such an extension preserves the semantics of FR, and therefore, the type safety claims of FR is preserved by dFR.

Note that the aim of this project is not to design a formal semantics or type system of Rust, rather, we aim to utilise existing work which provide formal semantics accompanied with a type system with type safety proofs for a core surface language of Rust (i.e., FR). Hence, here we only focus on discussing the semantics extension and the location transparency theorem. For readers who are curious about the (rather complicated) type system, please refer to appendix 3.A.

Term	$t ::=$	<code>let mut $x = t; t$</code>	declaration
		<code>$w := t$</code>	assignment
		<code>$t; t$</code>	sequence
		<code>()</code>	unit
		<code>{t}</code>	block
		<code>box t</code>	heap allocation
		<code>&w</code>	immutable borrow
		<code>&mut w</code>	mutable borrow
		<code>#w</code>	move
		<code>!w</code>	copy
		<code>v</code>	value
LVal	$w ::=$	<code>x</code>	variable
		<code>*w</code>	dereference
Value(\mathcal{V})	$v ::=$	<code>\perp</code>	
		<code>()</code>	unit
		<code>i</code>	integer
		<code>ℓ^\bullet</code>	owned reference
		<code>ℓ°</code>	borrowed reference
Location	$\ell \in \mathbb{A}$		

Figure 3.3: The revised syntax of FR

3.4.1 The Revised Syntax and Semantics of FR

We present a revised syntax of FR in figure 3.3. Note that `& w` and `&mut w` represent immutable and mutable borrowing, where `&[mut] w` represents either a immutable or a mutable borrow term. `ℓ^\bullet` and `ℓ°` are owned and borrowed references where `ℓ` represents a location in a program state. A copy term is denoted as `! w` . In addition, different from the original FR and Rust which do not have explicit syntax for move, we use `# w` to express move explicitly.

The notion of program state \mathcal{S} is introduced in figure 3.4, which is a mapping from a location to a tuple of value and lifetime. When a value is replaced or its lifetime is

$$\begin{aligned}
S &: \mathbb{A} \rightarrow \mathcal{V} \times \mathcal{L} \\
S \upharpoonright \ell &\mapsto (v, m) \quad \text{where: } \ell \in \mathbf{dom} S && \text{(update)} \\
S \otimes \ell &\mapsto (v, m) \quad \text{where: } \ell \notin \mathbf{dom} S && \text{(extend)}
\end{aligned}$$

Figure 3.4: The program state

expired, it will be removed from the program state.

We provide the operations of recursively removing values based on a location ℓ and a lifetime k from the state:

$$\begin{aligned}
(S \otimes (\ell^\bullet \mapsto (v, k))) \setminus \ell^\bullet &= S \setminus v \\
S \setminus v &= S \quad \text{otherwise}
\end{aligned}$$

and respectively:

$$S \setminus k(\ell) = \begin{cases} S(\ell) & \text{if } S(\ell) = (v, k) \\ \mathbf{undefined} & \text{otherwise} \end{cases}$$

Figure 3.5 shows the small-step operational semantics of FR, which takes the form of a reduction $S, t \xrightarrow{k} S', t'$, where S is the program state before the evaluation of the term t , and S', t' are the program state and the term after the evaluation. k is a lifetime, which we omit when it is irrelevant to the evaluation of a term.

Evaluating a copy term simply makes a copy of a value v at a given location ℓ , without modifying the program state, whereas evaluating a move term moves a value v out of a given location ℓ . A heap allocation box v puts the value v into a fresh location ℓ and gives it the *global lifetime* \top , which outlives all other lifetimes. The rule for evaluating a borrow term produces a borrowed reference of the give location ℓ . Assignment places a given value v' in the location ℓ , and recursively deallocates the old value v from the program state. Note that assignments to immutably borrowed references are prohibited by the type system provided by Pearce's (2021) original work, the discussion of the type system is omitted here since focus on the analysis of the SOS of FR programs. The evaluation of a declaration allocates a given value v to a fresh location ℓ and substitutes latter occurrence of the declared variable x with the owned reference ℓ^\bullet .

FR's lifetimes are based on the the lexical structure of programs. A block's lifetime k is based on the depth of the block. A block with deeper depth *suc* k lives shorter than a block with depth k . The evaluation of a block is the evaluation of the term inside

$$\begin{array}{c}
\frac{S(\ell) = (v, m)}{S, !\ell^\bullet \longrightarrow S, v} \text{ (COPY)} \qquad \frac{}{S \otimes \ell \mapsto (v, m), \# \ell^\bullet \longrightarrow S \otimes \ell \mapsto \perp, v} \text{ (MOVE)} \\
\\
\frac{\ell \notin \mathbf{dom} S}{S, \text{box } v \longrightarrow S \otimes \ell \mapsto (v, \top), \ell^\bullet} \text{ (BOX)} \qquad \frac{\ell \in \mathbf{dom} S}{S, \&[\text{mut}]\ell^\bullet \longrightarrow S, \ell^\circ} \text{ (BORROW)} \\
\\
\frac{}{S \otimes \ell \mapsto (v, m), \ell^\bullet := v' \longrightarrow (S \setminus v) \otimes \ell \mapsto (v', m), ()} \text{ (ASSIGN OWNED)} \\
\\
\frac{}{S \otimes \ell \mapsto (v, m), \ell^\circ := v' \longrightarrow (S \setminus v) \otimes \ell \mapsto (v', m), ()} \text{ (ASSIGN BORROWED)} \\
\\
\frac{\ell \notin \mathbf{dom} S}{S, \text{let mut } x = v; t \xrightarrow{k} S \otimes \ell \mapsto (v, k), t[\ell^\bullet/x]} \text{ (DECL)} \\
\\
\frac{}{S, \{v\} \xrightarrow{k} S \setminus \text{suc } k, v} \text{ (BLOCK (BASE))} \qquad \frac{S, t \xrightarrow{\text{suc } k} S', t'}{S, \{t\} \xrightarrow{k} S', \{t'\}} \text{ (BLOCK (SUC))} \\
\\
\frac{\ell \in \mathbf{dom} S}{S, \ell^\bullet; t \longrightarrow S \setminus \ell^\bullet, t} \text{ (SEQ-OWNEDREF)} \qquad \frac{\ell \in \mathbf{dom} S}{S, \ell^\circ; t \longrightarrow S, t} \text{ (SEQ-BORROWEDREF)} \\
\\
\frac{}{S, i; t \longrightarrow S, t} \text{ (SEQ-INT)} \qquad \frac{}{S, () ; t \longrightarrow S, t} \text{ (SEQ-UNIT)}
\end{array}$$

Figure 3.5: The semantics of revised FR

the block. At the end of the evaluation, a single value v is obtained and values that have short lifetime than the current block are deallocated from the program state. The reduction rules for the evaluation of sequences are intuitive. We highlight the case for a sequence of which the first term is an owned reference. After evaluating the owned reference, it is recursively deallocated from the program state.

An evaluation context is a term with a placeholder $[\cdot]$. $E[t]$ is a term obtained by replacing the placeholder with a term t . The evaluation context and reduction rule for the evaluation context are shown in figure 3.6.

Next, we will discuss the syntax and semantics of dFR, which extends FR with distributed computation features including remote references and values.

$$E ::= [\cdot] \mid E; t \mid v; E \mid \text{let mut } x = E; t \mid \text{let mut } x = v; E \mid \{E\} \mid \text{box } E \mid w = E$$

$$\frac{S, t \longrightarrow S, t'}{S, E[t] \longrightarrow S', E[t']} \text{ (CONTEXT)}$$

Figure 3.6: Evaluation context

3.4.2 The Syntax and Semantics of dFR

Figure 3.7 shows the syntax of dFR, which is the syntax of FR discussed in section 3.4.1 extended with the remote declaration $\text{let mut}@n x = t; t$, remote heap allocation $\text{box } t$, remote values $v@n$, and remote terms $t@n$. All extensions are highlighted. Note that n is the address of a node and \mathcal{N} is the set of addresses. Also, such an extension does not change the type system of FR.

Building upon the program state \mathcal{S} for FR, in figure 3.8, we introduce the distributed program state \mathcal{D} , which maps addresses of nodes to their program states. The reduction rule takes the form of $\mathcal{D}, C \longrightarrow \mathcal{D}', C'$, where C and C' are *configuration stacks*. For each reduction, the term on the top of a configuration stack C gets evaluated. The element of the configuration stack can either be a pair of an address and a hole $(n, ?)$ or a pair of an address and a term (n, t) . Utilising the distributed program state and the configuration stack, we provide the semantics of dFR in figure 3.9 and figure 3.10.

To evaluate copying a remotely owned reference $\ell^\bullet@n'$ on a node with address n , the first step shown in the rule COPY (s1) is to update the configuration stack by changing the $(n, !\ell^\bullet@n')$ to $(n, ?)$, and pushing a new address-term pair $(n', !\ell^\bullet)$ to be evaluated onto the stack. It models that the computation is passed to the node n' , which stores the resource owned by the reference $\ell^\bullet@n'$. Then the rule COPY (s2) indicates that the copy term $!\ell^\bullet$ gets evaluated on the node n' , and the resulting value annotated with the address n' is passed back to fill in the hole. Similarly, as for the semantics of moving the value out of a remote owned reference, the first step is to replace the term on the node n with a hole, and pass the copy term to the node n' to evaluate. The resulting value v annotated with the address n' is passed back to the node n at the end of the evaluation to fill in the hole, and the value of the location ℓ at the program state of the node n' is replaced by \perp , indicating that the value is moved out of the location ℓ at the node n' .

Term(\mathcal{T})	$t ::=$	$\text{let mut } x = t; t$	declaration
		$\text{let mut}@n x = t; t$	remote declaration
		$w := t$	assignment
		$t; t$	sequence
		$()$	unit
		$\{t\}$	block
		$\text{box } t$	heap allocation
		$\text{box}@n t$	remote heap allocation
		$\&w$	immutable borrow
		$\&\text{mut } w$	mutable borrow
		$\#w$	move
		$!w$	copy
		v	value
		$v@n$	remote value
Remote Term	$t_d ::=$	$t@n$	
LVal	$w ::=$	x	variable
		$*w$	dereference
Value(\mathcal{V})	$v ::=$	\perp	
		$()$	unit
		i	integer
		ℓ^\bullet	owned reference
		ℓ°	borrowed reference
Location	$\ell \in \mathbb{A}$	Node	$n \in \mathcal{N}$

Figure 3.7: The syntax of dFR

$$\begin{array}{ll}
\mathcal{D} : \mathcal{N} \rightarrow \mathcal{S} & nt \in \mathcal{N} \times \mathbb{1} + \mathcal{N} \times \mathcal{T} \\
C : (nt)^* & \text{Configuration} : \mathcal{D}, C
\end{array}$$

Figure 3.8: Distributed program state and configuration stack

$$\begin{array}{c}
\frac{\mathcal{D}(n')(\ell) = (v, m)}{\mathcal{D}, C \text{ ++ } (n, !\ell^\bullet @ n') \longrightarrow \mathcal{D}, C \text{ ++ } (n, ?) \text{ ++ } (n', !\ell^\bullet)} \text{ (COPY (s1))} \\
\\
\frac{\mathcal{D}(n')(\ell) = (v, m)}{\mathcal{D}, C \text{ ++ } (n, ?) \text{ ++ } (n', !\ell^\bullet) \longrightarrow \mathcal{D}, C \text{ ++ } (n, v @ n')} \text{ (COPY (s2))} \\
\\
\frac{}{\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \text{ ++ } (n, \# \ell^\bullet @ n') \longrightarrow \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \text{ ++ } (n, ?) \text{ ++ } (n', \# \ell^\bullet)} \text{ (MOVE (s1))} \\
\\
\frac{}{\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \text{ ++ } (n, ?) \text{ ++ } (n', \# \ell^\bullet) \longrightarrow \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto \perp), C \text{ ++ } (n, v @ n')} \text{ (MOVE (s2))} \\
\\
\frac{}{\mathcal{D}, C \text{ ++ } (n, (\text{box} @ n' v)) \longrightarrow \mathcal{D}, C \text{ ++ } (n, ?) \text{ ++ } (n', \text{box } v)} \text{ (BOX (s1))} \\
\\
\frac{\ell \notin \mathbf{dom } \mathcal{D}(n')}{\mathcal{D}, C \text{ ++ } (n, ?) \text{ ++ } (n', \text{box } v) \longrightarrow \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes \ell \mapsto (v, \top)), C \text{ ++ } (n, \ell^\bullet @ n')} \text{ (BOX (s2))} \\
\\
\frac{\ell \in \mathbf{dom } \mathcal{D}(n')}{\mathcal{D}, C \text{ ++ } (n, \&[\text{mut}] \ell^\bullet @ n') \longrightarrow \mathcal{D}, C \text{ ++ } (n, ?) \text{ ++ } (n', \&[\text{mut}] \ell^\bullet)} \text{ (BORROW (s1))} \\
\\
\frac{\ell \in \mathbf{dom } \mathcal{D}(n')}{\mathcal{D}, C \text{ ++ } (n, ?) \text{ ++ } (n', \&[\text{mut}] \ell^\bullet) \longrightarrow \mathcal{D}, C \text{ ++ } (n, \ell^\circ @ n')} \text{ (BORROW (s2))}
\end{array}$$

Figure 3.9: The semantics of dFR (part one)

$$\begin{array}{c}
\frac{}{\mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, m)), C \mathrel{++} (n, \ell @ n'^{\bullet} := v') \longrightarrow \mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, m)), C \mathrel{++} (n, ?) \mathrel{++} (n', \ell^{\bullet} := v')} \text{(ASSIGN OWNED (s1))} \\
\\
\frac{}{\mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, m)), C \mathrel{++} (n, ?) \mathrel{++} (n', \ell^{\bullet} := v') \longrightarrow \mathcal{D} \otimes (n' \mapsto S \setminus v \otimes \ell \mapsto (v', m)), C \mathrel{++} (n, ())} \text{(ASSIGN OWNED (s2))} \\
\\
\frac{}{\mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, m)), C \mathrel{++} (n, \ell @ n'^{\circ} := v') \longrightarrow \mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, m)), C \mathrel{++} (n, ?) \mathrel{++} (n', \ell^{\circ} := v')} \text{(ASSIGN BORROWED (s1))} \\
\\
\frac{}{\mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, m)), C \mathrel{++} (n, ?) \mathrel{++} (n', \ell^{\circ} := v') \longrightarrow \mathcal{D} \otimes (n' \mapsto S \setminus v \otimes \ell \mapsto (v', m)), C \mathrel{++} (n, ())} \text{(ASSIGN BORROWED (s2))} \\
\\
\frac{}{\mathcal{D}, C \mathrel{++} (n, \text{let mut } @n' x = v; t) \longrightarrow \mathcal{D}, C \mathrel{++} (n, t[?/x]) \mathrel{++} (n', \text{let mut } x = v; x)} \text{(DECL (s1))} \\
\\
\frac{\ell \notin \mathbf{dom} \mathcal{D}(n')}{\mathcal{D}, C \mathrel{++} (n, t[?/x]) \mathrel{++} (n', \text{let mut } x = v; x) \longrightarrow \mathcal{D} \otimes (n' \mapsto S \otimes \ell \mapsto (v, k)), C \mathrel{++} (n, t[\ell^{\bullet} @ n' / x])} \text{(DECL (s2))} \\
\\
\frac{S, t \longrightarrow S', t'}{\mathcal{D} \otimes (n \mapsto S), C \mathrel{++} (n, t) \longrightarrow \mathcal{D} \otimes (n \mapsto S'), C \mathrel{++} (n, t')} \text{(LOCAL TERMS)} \\
\\
\frac{\mathcal{D} \otimes (n' \mapsto S), C \mathrel{++} (n', t) \longrightarrow \mathcal{D} \otimes (n' \mapsto S'), C \mathrel{++} (n', t')}{\mathcal{D} \otimes (n' \mapsto S), C \mathrel{++} (n, t @ n') \longrightarrow \mathcal{D} \otimes (n' \mapsto S'), C \mathrel{++} (n, t' @ n')} \text{(REMOTE TERMS)}
\end{array}$$

Figure 3.10: The semantics of dFR (part two)

The rules Box (s1) and Box (s2) show the evaluation of a remote heap allocation $\text{box}@n' v$ on the node n . Firstly, the heap allocation is passed to the node n' to be evaluated, and a hole on the node n is created and pushed onto the configuration stack. The value v is stored in a fresh location ℓ at the program state of the node n' and assigned with the lifetime \top since it is a heap allocation, hence a owned reference ℓ^\bullet is created in the node n' . Such a owned reference is then passed back to the node n allowing the node n to own the location ℓ created by the heap allocation on the node n' . At the end of the evaluation, as shown in the rule Box (s2), on the top of the configuration stack, the hole create on the node n is filled by the owned remote reference $\ell^\bullet@n'$

The rules BORROW (s1) and BORROW (s2) show the evaluation of immutable and mutable borrow terms. On a given node n , to borrow a remotely owned reference from a different node n' , firstly a hole is created and pushed waiting for a term to be passed back, and the borrow term $\&[\text{mut}]\ell^\bullet@n'$ is passed to the node n' to be evaluated. After it being evaluated on the node n' , the resulting remotely borrowed reference $\ell^\circ@n'$ is passed back to the node n into the hole on the configuration stack.

A remote assignment to an owned reference $\ell^\bullet@n' := v'$ on the node n assigns a new value v' to its remotely owned reference on the node n' , which is shown in the rule ASSIGN OWNED (s1) and ASSIGN OWNED (s2). The first step is again leaving a hole awaiting to be filled on the configuration stack and passing the assignment to the node n' to be evaluated. In the next step, the evaluation of the assignment on the node n' modifies the program state on n' by recursively deallocates the old value v which is stored in the location ℓ . And then the location ℓ on the node n' is assigned with the new value v' . Since the assignment produces only a unit value $()$, it will be passed back and fill in the hole on the configuration stack. The evaluation of a remote assignment to a borrowed reference is similar. Note that again, since dFR extends FR without any modification of FR's type system, same to assignments in FR, remote assignments to immutable references in dFR are also prohibited by the type system.

Demonstrated in DECL (s1) and DECL (s2), the evaluation of a remote declaration is more complicated. The first step shown in DECL (s1) leaves a hole to be filled by the resulting term in the substitution of the declared variable x . Then the declaration is passed to the node n' to be evaluated. Shown in DECL (s2), once a fresh location ℓ on the node n' is allocated with the value v , the owned reference ℓ^\bullet will then be passed back to the node n and all occurrences of the declared variable x on the node n will be substituted with the remote owned reference $\ell^\bullet@n'$.

Lastly, all reduction rules for evaluating terms presented in FR are adapted into reduction rules for evaluating dFR via the rule LOCAL TERM. As for the evaluation of remote terms, shown in the rule REMOTE TERM, if a local term t on the node n' is evaluated into t' , then when it is treated as a remote term $t@n'$ on the node n , it will be evaluated to a remote term $t'@n'$ on the node n .

In the next section, we present and prove a location transparency theorem, which states that when translating a monolithic program written in FR into a distributed program written in dFR, the semantics of the monolithic program is preserved.

3.4.3 Preservation of Semantics when Translating a FR Program into a dFR Program

As we have mentioned in previous sections, by extending FR into dFR, the type system and static borrow checking of the validity of owning, immutably borrowing and mutably borrowing resources remain unchanged. By formalising the semantics of FR and dFR, we would like to show that, when we flatten a distributed program in dFR into a monolithic program in FR, the flattened result of the execution of the distributed program should be the same as the result of the execution of the flattened single node program. By demonstrating that the distributed program preserves the original semantics of the monolithic program, we can then conclude that the memory safety guarantees provided by FR's type system and static checking can be extended into distributed program in dFR.

Formally, we state this semantic preservation property of the distributed extension dFR in the *location transparency theorem* 3.4.1. Note that the reverse direction of the theorem does not hold. Since to extend a given single node program into a distributed program by allocating the program state on arbitrary nodes and making the term involved in the execution containing remote components that live on arbitrary nodes may not lead to constructing a distributed program that can be successfully executed. However, because our goal is to show that the distributed program preserves the same behaviour as if it is a single node program, having only one direction in the theorem is sufficient for our claim.

Theorem 3.4.1 (Location Transparency). For any term t , given an initial distributed program state \mathcal{D} and an initial single node program state \mathcal{S} , where the flattened distributed program state equals to the single node program state, if a distributed execution of a term t that may be a remote term or contain remote component with

the distributed program state \mathcal{D} results in a distributed program state \mathcal{D}' and a value v which can be either remote or local, then the execution of the flattened term t with the single node program state S will give a state S' and value v' , where the flattened resulting distributed program state $|\mathcal{D}'|$ equals to the S' and the flattened value v equals to v' . \longrightarrow^* is a reflexive transitive closure that represents one or more steps of evaluation. Note that we make locations on all nodes distinct to simplify the proof.

$$\begin{aligned} \forall t \in \mathcal{T}. \mathcal{D}, [(n, t)] &\longrightarrow^* \mathcal{D}', [(n, v)] \wedge |\mathcal{D}| = S \\ &\Rightarrow \\ S, t|_{@} &\longrightarrow S', v' \wedge |\mathcal{D}'| = S' \wedge v|_{@} = v' \end{aligned}$$

where $|\cdot|$ and $\cdot|_{@}$ are operators that erase addresses of nodes from distributed program states and terms defined as:

$$\begin{aligned} |\mathcal{D}| &= \bigcup_{\forall n \in \text{dom } \mathcal{D}} \mathcal{D}(n) \\ (\text{let mut } @n x = t; t)|_{@} &= \text{let mut } x = t; t & \text{box } @n t|_{@} &= \text{box } t \\ \ell^\bullet @n := v|_{@} &= \ell^\bullet := v & \ell^\circ @n := v|_{@} &= \ell^\circ := v \\ v @n|_{@} &= v & t @n|_{@} &= t|_{@} \end{aligned}$$

Since a term t is always a closed term, we prove the theorem 3.4.1 by structural induction on the term t . We focus on presenting proofs of the cases concerning the remote extensions in dFR's reduction rules given in figure 3.9 and figure 3.10.

Case COPY. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \quad S = S_0 \otimes (\ell \mapsto (v, m)) \quad |\mathcal{D}_0| = S_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = S$$

Following the dFR's reduction rule COPY (s1) and COPY (s2), the distributed execution of a remote copy term gives:

$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), [(n, !\ell^\bullet @n')] \longrightarrow^* \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), [(n, v @n')]$$

Following FR's reduction rule COPY, the monolithic execution of the flattened remote copy term gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), !\ell^\bullet @ n' |_{@} \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto (v, m)), v$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \quad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto (v, m))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program state are equal, and the flattened resulting value obtained from the distributed execution and the value obtained from the monolithic execution are equal:

$$|\mathcal{D}'| = \mathcal{S}' \quad v @ n' |_{@} = v$$

Hence:

$$\begin{aligned} \mathcal{D}, [(n, !\ell^\bullet @ n')] &\longrightarrow^* \mathcal{D}', [(n, v @ n')] \wedge |\mathcal{D}| = \mathcal{S} \\ &\Rightarrow \\ \mathcal{S}, !\ell^\bullet @ n' |_{@} &\longrightarrow \mathcal{S}', v \wedge |\mathcal{D}'| = \mathcal{S}' \wedge v @ n' |_{@} = v \end{aligned}$$

□

Case MOVE. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \quad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \quad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule MOVE (s1) and MOVE (s2), the distributed execution of a remote move term gives:

$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), [(n, \# \ell^\bullet @ n')] \longrightarrow^* \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto \perp)), [(n, v @ n')]$$

Following FR's reduction rule MOVE, the monolithic execution of the flattened remote move term gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), \# \ell^\bullet @ n' |_{@} \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto \perp), v$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto \perp)) \quad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto \perp)$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the flattened resulting value obtained from the distributed execution and the resulting value obtained from the monolithic execution are equal:

$$|\mathcal{D}'| = |\mathcal{D}_0| \otimes (\ell \mapsto \perp) = \mathcal{S}_0 \otimes (\ell \mapsto \perp) = \mathcal{S}' \quad v@n'|_{@} = v$$

Hence:

$$\begin{aligned} \mathcal{D}, [(n, \# \ell^\bullet @ n')] &\longrightarrow^* \mathcal{D}', [(n, v@n')] \wedge |\mathcal{D}| = \mathcal{S} \\ &\Rightarrow \\ \mathcal{S}, \# \ell^\bullet @ n'|_{@} &\longrightarrow \mathcal{S}', v \wedge |\mathcal{D}'| = \mathcal{S}' \wedge v@n'|_{@} = v \end{aligned}$$

□

Case Box. We assume that before the evaluation, given the distributed program state \mathcal{D} and monolithic program state \mathcal{S} where the flatten distributed program state equals to the monolithic program state. In addition, we take a fresh location ℓ :

$$|\mathcal{D}| = \mathcal{S} \quad \ell \notin \text{dom } \mathcal{D}(n') \quad \ell \notin \text{dom } \mathcal{S}$$

Following the dFR's reduction rule Box (s1) and Box (s2), the distributed execution of a remote heap allocation term gives:

$$\mathcal{D}, [(n, \text{box}@n' v)] \longrightarrow^* \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, \top))), [(n, \ell^\bullet @ n')]$$

Following FR's reduction rule Box, the monolithic execution of the flattened remote heap allocation term gives:

$$\mathcal{S}, (\text{box}@n' v)|_{@} \longrightarrow \mathcal{S} \otimes (\ell \mapsto (v, \top)), \ell^\bullet$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, \top))) \quad \mathcal{S}' = \mathcal{S} \otimes (\ell \mapsto (v, \top))$$

The updated distributed program state can be flattened into:

$$|\mathcal{D}'| = |\mathcal{D}| \otimes (\ell \mapsto (v, \top))$$

Since we have $|\mathcal{D}| = \mathcal{S}$, we can conclude that the flattened updated distributed program state and the updated monolithic program state are equal:

$$|\mathcal{D}'| = |\mathcal{D}| \otimes (\ell \mapsto (v, \top)) = \mathcal{S} \otimes (\ell \mapsto (v, \top)) = \mathcal{S}'$$

Also, the flattened remote owned reference obtained from the distributed execution equals to the owned reference obtained from the monolithic execution:

$$\ell^\bullet @ n' |_@ = \ell^\bullet$$

Hence:

$$\begin{aligned} \mathcal{D}, [(n, \text{box}@n' v)] &\longrightarrow^* \mathcal{D}', [(n, \ell^\bullet)] \wedge |\mathcal{D}| = \mathcal{S} \\ &\Rightarrow \\ \mathcal{S}, (\text{box}@n' v) |_@ &\longrightarrow \mathcal{S}', \ell^\bullet \wedge |\mathcal{D}'| = \mathcal{S}' \wedge \ell^\bullet @ n' |_@ = \ell^\bullet \end{aligned}$$

□

Case BORROW. We assume that before the evaluation, given the distributed program state \mathcal{D} and monolithic program state \mathcal{S} where the flatten distributed program state equals to the monolithic program state. In addition, ℓ is a location with an allocated value:

$$|\mathcal{D}| = \mathcal{S} \quad \ell \in \mathbf{dom} \mathcal{D}(n') \quad \ell \in \mathbf{dom} \mathcal{S}$$

Following the dFR's reduction rule BORROW (s1) and BORROW (s2), the distributed execution of a remote borrow term gives:

$$\mathcal{D}, [(n, \&[\text{mut}] \ell^\bullet @ n')] \longrightarrow^* \mathcal{D}, [(n, \ell^\circ @ n')]$$

Following FR's reduction rule BORROW, the monolithic execution of the flattened remote borrow term gives:

$$\mathcal{S}, (\&[\text{mut}] \ell^\bullet @ n') |_@ \longrightarrow \mathcal{S}, \ell^\circ$$

To conclude, after the evaluation, the distributed program state and the monolithic program state remain unchanged, hence they are still equal. In addition, the flattened remote borrowed reference obtained from the distributed execution equals to the borrowed reference obtained from the monolithic execution:

$$\ell^\circ @ n' |_@ = \ell^\circ$$

Hence:

$$\begin{aligned}
\mathcal{D}, [(n, \&[\text{mut}]\ell^\bullet @ n')] &\longrightarrow^* \mathcal{D}, [(n, \ell^\circ)] \wedge |\mathcal{D}| = \mathcal{S} \\
&\Rightarrow \\
\mathcal{S}, (\&[\text{mut}]\ell^\bullet @ n')|_{@} &\longrightarrow \mathcal{S}, \ell^\circ \wedge |\mathcal{D}| = \mathcal{S} \wedge \ell^\circ @ n'|_{@} = \ell^\circ
\end{aligned}$$

□

Case ASSIGN OWNED. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \quad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \quad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule ASSIGN OWNED (s1) and ASSIGN OWNED (s2), the distributed execution of a remote assignment to an owned reference gives:

$$\begin{aligned}
\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), [(n, \ell^\bullet @ n' := v')] & \\
\longrightarrow^* & \\
\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))), [(n, () @ n')] &
\end{aligned}$$

Following FR's reduction rule ASSIGN OWNED, the monolithic execution of the flattened remote assignment to an owned reference gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), (\ell^\bullet @ n' := v')|_{@} \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto (v', m)), ()$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))) \quad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto (v', m))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the flattened unit value obtained from the distributed execution and the unit value obtained from the monolithic execution are trivially equal:

$$|\mathcal{D}'| = |\mathcal{D}_0| \otimes (\ell \mapsto (v', m)) = \mathcal{S}_0 \otimes (\ell \mapsto (v', m)) = \mathcal{S}' \quad () @ n'|_{@} = ()$$

Hence:

$$\begin{aligned}
\mathcal{D}, [(n, \ell^\bullet @ n' := v')] &\longrightarrow^* \mathcal{D}', [(n, () @ n')] \wedge |\mathcal{D}| = \mathcal{S} \\
&\Rightarrow \\
\mathcal{S}, (\ell^\bullet @ n' := v')|_{@} &\longrightarrow \mathcal{S}', () \wedge |\mathcal{D}'| = \mathcal{S}' \wedge () @ n'|_{@} = ()
\end{aligned}$$

□

Case ASSIGN BORROWED. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \quad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \quad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule *ASSIGN BORROWED* (s1) and *ASSIGN BORROWED* (s2), the distributed execution of a remote assignment to a (mutably) borrowed reference gives:

$$\begin{aligned}
&\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), [(n, \ell^\circ @ n' := v')] \\
&\quad \longrightarrow^* \\
&\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))), [(n, () @ n')]
\end{aligned}$$

Following FR's reduction rule *ASSIGN BORROWED*, the monolithic execution of the flattened remote assignment to a (mutably) borrowed reference gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), (\ell^\circ @ n' := v')|_{@} \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto (v', m)), ()$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))) \quad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto (v', m))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the flattened unit value obtained from the distributed execution and the unit value obtained from the monolithic execution are trivially equal:

$$|\mathcal{D}'| = |\mathcal{D}_0| \otimes (\ell \mapsto (v', m)) = \mathcal{S}_0 \otimes (\ell \mapsto (v', m)) = \mathcal{S}' \quad () @ n'|_{@} = ()$$

Hence:

$$\begin{aligned}
\mathcal{D}, [(n, \ell^\circ @ n' := v')] &\longrightarrow^* \mathcal{D}', [(n, () @ n')] \wedge |\mathcal{D}| = \mathcal{S} \\
&\Rightarrow \\
\mathcal{S}, (\ell^\circ @ n' := v')|_{@} &\longrightarrow \mathcal{S}', () \wedge |\mathcal{D}'| = \mathcal{S}' \wedge () @ n'|_{@} = ()
\end{aligned}$$

□

Case DECL. We assume that before the evaluation, given the distributed program state \mathcal{D} and monolithic program state \mathcal{S} where the flattened distributed program state equals to the monolithic program state. In addition, we take a fresh location ℓ :

$$|\mathcal{D}| = \mathcal{S} \quad \ell \notin \mathbf{dom} \mathcal{D}(n') \quad \ell \notin \mathbf{dom} \mathcal{S}$$

Following the dFR's reduction rule DECL (s1) and DECL (s2), the distributed execution of a remote declaration gives:

$$\mathcal{D}, [(n, \text{let mut}@n' x = v; t)] \longrightarrow^* \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, k))), [(n, t[\ell^\bullet @ n' / x])]$$

Following FR's reduction rule DECL, the monolithic execution of the flattened remote declaration gives:

$$\mathcal{S}, (\text{let mut}@n' x = v; t)|_{@} \longrightarrow \mathcal{S} \otimes (\ell \mapsto (v, k)), t[\ell^\bullet / x]$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, k))) \quad \mathcal{S}' = \mathcal{S} \otimes (\ell \mapsto (v, k))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the substitution with the flattened owned reference obtained from the distributed execution and the substitution with the owned reference obtained from the monolithic execution are trivially equal:

$$\begin{aligned}
|\mathcal{D}'| &= |\mathcal{D}| \otimes (\ell \mapsto (v, k)) = \mathcal{S} \otimes (\ell \mapsto (v, k)) = \mathcal{S}' \\
t[\ell^\bullet @ n' / x]|_{@} &= t[(\ell^\bullet @ n')|_{@} / x] = t[\ell^\bullet / x]
\end{aligned}$$

Hence:

$$\begin{aligned}
\mathcal{D}, [(n, \text{let mut}@n' x = v; t)] &\longrightarrow^* \mathcal{D}', [(n, t[\ell^\bullet @ n' / x])] \wedge |\mathcal{D}| = \mathcal{S} \\
&\Rightarrow \\
\mathcal{S}, (\text{let mut}@n' x = v; t)|_{@} &\longrightarrow \mathcal{S}', t[\ell^\bullet / x] \wedge |\mathcal{D}'| = \mathcal{S}' \wedge t[\ell^\bullet @ n' / x]|_{@} = t[\ell^\bullet / x]
\end{aligned}$$

□

The proofs for location transparency of the distributed and monolithic executions of local terms and remote terms should then be trivial.

3.4.4 Summary

In section 3.3, we have discussed the design and implementation of a UMI framework as a library in Rust. The core designed concepts of such a library — extending Rust’s memory safety guarantees into a distributed setting — is presented in the formalisation of a distributed extension of a core calculus of Rust in this section. By proving the location transparency theorem 3.4.1, we demonstrate that a distributed program developed using the UMI framework preserves the semantics of a monolithic program from which it is translated. Therefore, with our UMI framework, Rust’s memory safety guarantees provided by its type system, lifetime and ownership system, and borrow checking mechanism are indeed extended into a distributed setting.

3.5 Related Work

Design and Implementations of Remote Procedural Call Described in Nelson’s (1981) thesis, RPC allows programs in separate address spaces to communicate synchronously. By experimenting with different implementations of RPC, Nelson (1981) argues that RPC is an efficient and effective programming tool for distributed systems. Java RMI (Wollrath et al., 1996) implements the concept of RPC in a object-orientated programming language. It outlines a model for distributed objects within the Java environment which allows Java objects to communicate across different address spaces. This RMI framework is designed to integrate seamless with the Java language, preserving as much of the Java object model’s semantics as possible. As for its memory management mechanism, it contains a design of distributed garbage collection, ensuring that remote objects which are no longer referenced by any client should be automatically garbage collected. However, the Stub object in this framework does not always preserves the semantics of the Java object model. In our UMI approach, the framework is designed to be truly integrated with Rust as it is semantic preserving. The idea of location transparency and seamless integration of distributed computing presented in this UMI framework is related on the formal model Krivine Nets presented by Fredriksson and Ghica (2014), which extends the classic Krivine abstract machine to support distributed execution. This extension allows a seamless and

transparent RPC mechanism to handle higher-order functions without transmitting the actual code. Krivine Nets enable the seamless integration of distributed computing into programming languages by eliminating explicit communication and process management from source code. In addition, instead of embedding deployment details within the source code, Krivine Nets handle these details automatically. The RPC calculus presented by Cooper and Wadler (2009) explores the design and implementation of a symmetrical location-aware programming language atop a stateless server. The authors address the challenge of maintaining control state transparently within a programming language, despite the stateless nature of web servers, which typically do not retain client-specific session information. To achieve this, they propose a technique involving three main steps: defunctionalisation, continuation-passing style (CPS) translation, and the use of a trampoline mechanism for server-to-client requests. The paper outlines the RPC calculus (λ_{rpc}), which is enriched with location annotations to indicate where code should execute, supporting semantics where computation steps can occur at designated locations; and a translation from λ_{rpc} to a first-order client-server calculus (λ_{cs}), which models an asymmetrical client-server environment. They demonstrate that this translation preserves the locative semantics of the source calculus, allowing for effective RPC calls in the location-aware language. The subsequent work for such a RPC calculus present by (?) proposes a polymorphic RPC (Remote Procedure Call) calculus that extends the typed RPC calculus with polymorphic locations. It introduces a new polymorphic RPC calculus that allows programmers to write succinct multi-tier programs using polymorphic location constructs and defines a type system for the polymorphic RPC calculus and proves its type soundness. In addition it develops a monomorphisation translation that converts polymorphic RPC terms into monomorphic typed RPC terms, allowing existing slicing compilation methods for client-server models to be used. The type correctness and semantic correctness of the monomorphisation translation are proven.

Distributed Memory Management Distributed garbage collection used in Java RMI (Wollrath et al., 1996) as been an essential yet challenging research topic in distributed memory management. Abdullahi and Ringwood (1998) offers a comprehensive review of distributed garbage collection (GC) schemes applicable to autonomous systems connected by a network, particularly in the context of Internet programming languages such as Java. It highlights the evolution of garbage collection from single-address-space collectors to distributed systems due to the increasing promi-

nence of languages like Java in Internet applications. It categorises and reviews various GC methods, discussing their adaptation for distributed environments, which are characterised by issues such as communication overhead, locality of action, and non-deterministic communication latency. Designed for the object-oriented programming language with actor model Pony (Clebsch et al., 2017), Orca is a concurrent garbage collection algorithm which manages memory without requiring stop-the-world pauses or synchronisation mechanisms, enabling zero-copy message passing and mutable data sharing among actors. It leverages Pony’s type system to ensure data race-free programs, allowing garbage collection tasks to be handled locally by each actor without synchronisation, thus enhancing performance and responsiveness. Perhaps more relevant to our memory management mechanism, the region system Reggio (Arvidsson et al., 2023) accompanied with a type system for Verona, which is a concurrent object-oriented programming language, organises objects into isolated regions, each with its own memory management strategy. It addresses the challenge of providing the control of manual memory management while maintaining memory safety by utilising a combination of region-based memory partitioning and an ownership type system.

Formalisations of Rust and Their Limitations The formalism of a core calculus of the UMI framework presented in this chapter is based on a formalisation of a core calculus of Rust described by Pearce (2021). Although as mentioned in previous sections, we are not completely satisfied with this formalism, it is the most suitable choice comparing to all common approaches of formalisation of Rust’s semantics we have surveyed. In the following paragraphs, we discuss these approaches in detail.

There are many different approaches to formalise Rust, from different perspectives and aiming for different application domains. RustBelt (Jung et al., 2017b) provide a formalised continuation-passing style $\text{MIR} - \lambda_{\text{Rust}}$ — mechanised using Iris. However, such an approach does not provide a formal model which is close to the source-level language of Rust. Hence, it is not convenient for us to use it as the basis for formalising the distributed extensions of the features of the UMI framework.

Different from RustBelt, there are also attempts of formalising Rust from a source-level language perspective. For instance, Oxide (Weiss et al., 2021) attempts to formalise near-complete source-level Rust language features, providing a type system and small-step operational semantics. An implementation of Oxide is provided on GitHub although the formal claims are not mechanised. We argue that due to the

level of the complexity and obscurity of this formalism, it is rather hard for us to gain a clear understanding of the modelling of Rust’s borrow checking and non-lexical lifetime. In addition, without a concise and consistent presentation of the syntax, type system and formal semantics of the core features of Rust, it is again hard for us to use such a formalism to reason about properties of Rust programs and be convinced that the type system is indeed sound.

Back to the formalisation we choose to build our distributed language extension upon, instead of modelling the full Rust language, Pearce (2021) formalises a core calculus of Rust, which is FR, emphasising on the understanding of borrowing and lifetime of Rust. However, this formalism does not include essential Rust language features such as tuples, structs, functions and closures which makes it “too minimalist”. One may argue that Pearce (2021) does discuss the possible extensions of FR to include tuples and functions hence it is unfair to criticise such an approach being too minimalist, however, I insist that these features are fundamental building blocks of a core language and should not be treated as extensions of a core language. Besides, the way that the let-binding is modelled made it hard to do substitutions, which is in my opinion the obstacle of having functions as a part of the formalism. In addition, its lexical treatment of Rust’s lifetime is already obsolete.

Due to the reality of lack of a concise formalism of the source-level Rust that captures all key concepts of Rust’s language features, our formalism of the UMI framework as a distributed extension of Rust is also not completely satisfying. However, it does demonstrate Rust’s core memory management mechanisms, and allows us to prove the semantic preservation property of distributed programs implemented using the UMI framework. To have formalism of UMI capturing more language features would require us to develop yet another formalism of source-level Rust, which is out of the scope of this project.

3.6 Conclusion

In this chapter, we firstly present our design, implementation, and formalisation of a UMI framework for Rust. This UMI framework allows programmers to express distributed computation in the same form of monolithic computation, abstracting away the internet communication complications and message passing details. To demonstrate as a distributed extension of Rust, our UMI framework extends Rust’s memory safety guarantees into a distributed setting, we present the formalism of a core

calculus focusing on the core features relating to distributed memory management mechanisms in the UMI framework. We present FR, which is a core calculus of the surface language of Rust, formalising the key concepts of Rust’s ownership and lifetime in memory management. Then we extend FR into dFR, to include distributed features of the UMI framework. By showing that distributed programs written in dFR preserves the semantics of monolithic programs written in FR via proving a location transparency theorem, we can conclude that the memory safety guarantees provided by Rust can be extended to distributed programs written with our UMI framework.

Epilogue

In this study, we have designed and implemented a distributed computing framework in Rust which allows distributed programs to preserve the semantics of monolithic programs. Such a design makes it easier for programmers to migrate monolithic Rust programs into distributed setting while adopting the memory safety guarantees of these monolithic programs. In the end, I would like to enclose this chapter with some high-level discussions.

Our UMI framework provides a new perspective for characterising the relationship between a monolithic program and a distributed program — a monolithic program can be viewed as an abstraction of a distributed program. It specifies the intend functionalities to be achieved by a distributed program while encapsulating the detailed implementations message passing, data serialisation and deserialisation, and other network communication complications.

However, with such an abstract view of distributed program, network communication errors such as timeouts and server errors are not explicitly modelled or handled with the UMI framework. Although we envision integrating such a framework within micro-services platforms where server errors are mitigated by cloud service providers, and supervision strategies can be used for taking snapshots and restarting from a failure, making these issues not a critical problem of the design of the UMI framework, we have to admit that, such a framework itself is not expressive enough to model network communication errors.

One observation is that there is a trade-off between abstraction and expressiveness in modelling a distributed system. In this study, by making external facilities for handling the network communication issues, we focus on designing a concise abstraction which describes the intended functionalities of distributed programs and making it easier to model and reason about memory safety of distributed programs.

Partial Types	$\tilde{T} ::= T$	type
	$\Box \tilde{T}$	partial box
	$[T]$	undefined
Types	$T ::= \epsilon$	unit
	<code>int</code>	integer
	$\&\text{mut } \bar{w}$	mutable borrow
	$\&\bar{w}$	immutable borrow
	$\Box T$	box

Figure 3.11: Syntax of Types

3.A The Type System of FR

Figure 3.11 presents the syntax of types of Pearce’s (2021) FR without any modification. A primitive type such as the integer type `int` has copy semantics. A box type $\Box T$ that represents a heap allocation has move semantics. A partial type may contain *undefined* components. An undefined component denoted by $[T]$ represents a currently inaccessible location as its value has already been moved. The dFR is merely a semantics extension of FR, which uses the same type system as FR.

3.A.1 Preliminaries

These are support functions presented by Pearce (2021) for defining the typing rules.

Definition 3.A.1 (Copy Types). *A type T has copy semantics, denoted by $\text{copy}(T)$, when $T = \text{int}$ or $T = \&\bar{w}$.*

Note that mutable references and boxes do not have copy semantics.

Definition 3.A.2 (Type Strengthening). *For partial types \tilde{T}_1 and \tilde{T}_2 , we say that \tilde{T}_1 strengthens \tilde{T}_2 , denoted by $\tilde{T}_1 \sqsubseteq \tilde{T}_2$, according to the following rules:*

$$\begin{array}{c}
\frac{}{\tilde{T}_1 \sqsubseteq \tilde{T}_1} \text{ (W-REFLEX)} \qquad \frac{\tilde{T}_1 \sqsubseteq \tilde{T}_2}{\Box \tilde{T}_1 \sqsubseteq \Box \tilde{T}_2} \text{ (W-BOX)} \qquad \frac{u \sqsubseteq w}{\Gamma \vdash \&[\text{mut}] u \sqsubseteq \&[\text{mut}] w} \text{ (W-BOR)} \\
\\
\frac{T_1 \sqsubseteq T_2}{[T_1] \sqsubseteq [T_2]} \text{ (W-UNDEFA)} \qquad \frac{T_1 \sqsubseteq T_2}{T_1 \sqsubseteq [T_2]} \text{ (W-UNDEFB)} \qquad \frac{\tilde{T}_1 \sqsubseteq [T_2]}{\Box \tilde{T}_1 \sqsubseteq \Box [T_2]} \text{ (W-UNDEFC)}
\end{array}$$

Note that the rule W-BOR requires the mutability to be the same on both sides.

Definition 3.A.3 (Type Join). *The join of partial types \widetilde{T}_1 and \widetilde{T}_2 , denoted $\widetilde{T}_1 \sqcup \widetilde{T}_2$, is a partial function returning the strongest \widetilde{T}_3 such that $\widetilde{T}_1 \sqsubseteq \widetilde{T}_3$ and $\widetilde{T}_2 \sqsubseteq \widetilde{T}_3$.*

Definition 3.A.4 (Environment Strengthening). *Let Γ_1 and Γ_2 be typing environments. We say that Γ_1 strengthens Γ_2 , denoted $\Gamma_1 \sqsubseteq \Gamma_2$, if and only if $\text{dom}(\Gamma_1) = \text{dom}(\Gamma_2)$ and for all $x \in \text{dom}(\Gamma_1)$ where $\Gamma_1(x) = \langle \widetilde{T}_1 \rangle^l$, we have $\Gamma_2(x) = \langle \widetilde{T}_2 \rangle^l$ where $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$.*

Definition 3.A.5 (Environment Join). *The join of environments Γ_1 and Γ_2 , denoted $\Gamma_1 \sqcup \Gamma_2$, is a partial function returning the strongest Γ_3 such that $\Gamma_1 \sqsubseteq \Gamma_3$ and $\Gamma_2 \sqsubseteq \Gamma_3$.*

Definition 3.A.6 (LVal Typing). *An lval w is said to be typed with respect to an environment Γ , denoted $\Gamma \vdash w : \langle \widetilde{T} \rangle$, according to the following rules:*

$$\frac{\Gamma(x) = \langle \widetilde{T} \rangle^m}{\Gamma \vdash x : \langle \widetilde{T} \rangle^m} \text{ (T-LvVAR)} \qquad \frac{\Gamma \vdash w : \langle \square \widetilde{T} \rangle^m}{\Gamma \vdash *w : \langle \widetilde{T} \rangle^m} \text{ (T-LvBOX)}$$

$$\frac{\Gamma \vdash w : \langle \&[\text{mut}] \bar{u} \rangle^n \quad \Gamma \vdash u : \langle T \rangle^m}{\Gamma \vdash *w : \langle \bigsqcup_i T_i \rangle^{\prod_i m_i}} \text{ (T-LvBOR)}$$

Definition 3.A.7 (Path). *A path π is a sequence of zero or more path selectors ρ , which is either empty ($\pi \triangleq \epsilon$) or composed by appending a selector onto another path ($\pi \triangleq \pi' \cdot \rho$).*

Definition 3.A.8 (Path Selector). *A path selector ρ is always a dereference ($\rho \triangleq *$).*

Definition 3.A.9 (Path Conflict). *Let $u \triangleq \pi_u \mid x$ and $w \triangleq \pi_w \mid y$ be lvals. Then w is said to conflict with u , denoted $u \bowtie w$, if $x = y$.*

Note that $u \triangleq \pi \mid x$ denotes *destructuring* of an lval u into its base x and path π .

Definition 3.A.10 (Type Containment). *Let Γ be an environment where $\Gamma(x) = \langle \widetilde{T} \rangle^l$ for some l . Then $\Gamma \vdash x \rightsquigarrow T_y$ denotes that variable x contains type T_y and is defined as $\text{contains}(\Gamma, \widetilde{T}, T_y)$ where:*

$$\text{contains}(\Gamma, \widetilde{T}, T_y) = \begin{cases} \text{contains}(\Gamma, \square \widetilde{T}', T_y) & \text{if } \widetilde{T} = \square \widetilde{T}', \\ \text{true} & \text{if } \widetilde{T} = T_y, \\ \text{false} & \text{otherwise} \end{cases}$$

Definition 3.A.11 (Read Prohibited). *In an environment Γ , an lval w is said to be read prohibited, denoted $\text{readProhibited}(\Gamma, w)$, when some x exists where $\Gamma \vdash x \rightsquigarrow \&\text{mut } \bar{u}$ and $\exists_i (u_i \bowtie w)$.*

Definition 3.A.12 (Write Prohibited). *In an environment Γ , an lval w is said to be write prohibited, denoted $\text{writeProhibited}(\Gamma, w)$, when either some x exists where $\Gamma \vdash x \rightsquigarrow \&\bar{u} \wedge \exists_i (u_i \bowtie w)$ or $\text{readProhibited}(\Gamma, w)$ holds.*

The partial function $\text{move}(\Gamma, w)$ determines the environment after the value of an lval w is moved out:

Definition 3.A.13 (Move). *Let Γ be an environment where $\Gamma(x) = \langle \tilde{T}_1 \rangle^l$ for some lifetime l , and w an lval where $w \triangleq \pi_x \mid x$. Then $\text{move}(\Gamma, w)$ is a partial function defined as $\Gamma[x \mapsto \langle \tilde{T}_2 \rangle^l]$ where $\tilde{T}_2 = \text{strike}(\pi_x \mid \tilde{T}_1)$:*

$$\begin{aligned} \text{strike}(\epsilon \mid T) &= [T] \\ \text{strike}((\pi \cdot *) \mid \square \tilde{T}_1) &= \square \tilde{T}_2 \quad \textbf{where } \tilde{T}_2 = \text{strike}(\pi \mid \tilde{T}_1) \end{aligned}$$

Definition 3.A.14 (Mutable). *Let Γ be an environment where $\Gamma(x) = \langle \tilde{T} \rangle^l$ for some lifetime l , and w an lval where $w \triangleq \pi_x \mid x$. Then $\text{mut}(\Gamma, w)$ is a partial function defined as $\text{mutable}(\Gamma, \pi_x \mid \tilde{T})$ that determines whether w is mutable:*

$$\begin{aligned} \text{mutable}(\Gamma, \epsilon \mid T) &= \text{true} \\ \text{mutable}(\Gamma, (\pi \cdot *) \mid \square T) &= \text{mutable}(\Gamma, \pi \mid T) \\ \text{mutable}(\Gamma, (\pi \cdot *) \mid \&\text{mut } \bar{w}) &= \bigwedge_i \text{mut}(\Gamma, \pi \cdot w_i) \end{aligned}$$

Definition 3.A.15 (Environment Drop). *The environment drop deallocates locations by removing them from an environment Γ : $\text{drop}(\Gamma, m) = \Gamma - \{x \mapsto \langle \tilde{T} \rangle^m \mid x \mapsto \langle \tilde{T} \rangle^m \in \Gamma\}$.*

Definition 3.A.16 (Well-formed Type). *For an environment Γ , a type T is said to be well-formed with respect to a lifetime l , denoted $\Gamma \vdash T \geq l$, according to rules:*

$$\begin{array}{ccc} \frac{}{\Gamma \vdash \text{int} \geq l} \text{ (L-INT)} & \frac{\Gamma \vdash u : \langle T \rangle^m \quad m \geq l}{\Gamma \vdash \&[\text{mut}] u \geq l} \text{ (L-BORROW)} & \frac{\Gamma \vdash T \geq l}{\Gamma \vdash \square T \geq l} \text{ (L-BOX)} \end{array}$$

Definition 3.A.17 (Compatible Shape). *For an environment Γ , two partial types \tilde{T}_1 and \tilde{T}_2 are shape compatible, denoted as $\Gamma \vdash \tilde{T}_1 \approx \tilde{T}_2$, according to the following rules:*

$$\begin{array}{c}
\frac{}{\Gamma \vdash \text{int} \approx \text{int}} \text{ (S-INT)} \qquad \frac{\Gamma \vdash \widetilde{T}_1 \approx \widetilde{T}_2}{\Gamma \vdash \Box \widetilde{T}_1 \approx \Box \widetilde{T}_2} \text{ (S-Box)} \\
\\
\frac{\forall_{i,j} (\Gamma \vdash u_i : \widetilde{T}_1 \approx \widetilde{T}_2 : w_j \dashv \Gamma)}{\Gamma \vdash \&[\text{mut}] \bar{u} \approx \&[\text{mut}] \bar{w}} \text{ (S-BOR)} \qquad \frac{\Gamma \vdash T_1 \approx \widetilde{T}_2}{\Gamma \vdash \lfloor T_1 \rfloor \approx \widetilde{T}_2} \text{ (S-UNDEF L)} \\
\\
\frac{\Gamma \vdash \widetilde{T}_1 \approx T_2}{\Gamma \vdash \widetilde{T}_1 \approx \lfloor T_2 \rfloor} \text{ (S-UNDEF R)}
\end{array}$$

Definition 3.A.18 (Write). Let Γ be an environment where $\Gamma(x) = \langle \widetilde{T}_1 \rangle^l$ for some lifetime l and lval w where $w \triangleq \pi_x \mid x$. Then, $\text{write}^k(\Gamma, w, T)$ is a partial function defined as $\Gamma_2[x \mapsto \langle \widetilde{T}_2 \rangle^l]$ for some rank $k \geq 0$ where $(\Gamma_2, \widetilde{T}_2) = \text{update}^k(\Gamma, \pi_x \mid \widetilde{T}_1, T)$:

$$\begin{aligned}
& \text{update}^0(\Gamma, \epsilon \mid \widetilde{T}_1, T_2) = (\Gamma, T_2) \\
& \text{update}^{k \geq 1}(\Gamma, \epsilon \mid T_1, T_2) = (\Gamma, T_1 \sqcup T_2) \\
& \text{update}^k(\Gamma_1, (\pi \cdot *) \mid \Box \widetilde{T}_1, T) = (\Gamma_2, \Box \widetilde{T}_2) \quad \textbf{where } (\Gamma_2, \widetilde{T}_2) = \text{update}^k(\Gamma_1, \pi \mid \widetilde{T}_1, T) \\
& \text{update}^k(\Gamma, (\pi \cdot *) \mid \&\text{mut } \bar{u}_i, T) = (\bigsqcup_i \Gamma_i, \&\text{mut } \bar{u}_i) \quad \textbf{where } \Gamma_i = \text{write}^{k+1}(\Gamma, \pi \mid u_i, T)
\end{aligned}$$

3.A.2 Typing Rules

Utilising the helper functions defined in perious section, FR's typing rules are shown in figure 3.12. Note that we do not make any major modification to these typing rules. The only minor changes are: 1) The syntax of the lifetime of a block is now implicit; and 2) Since the syntax of declaration is changed from the original form $\text{let mut } x = t$ to $\text{let mut } x = t_1; t_2$ to allow substitutions, the typing rule is slightly modified accordingly.

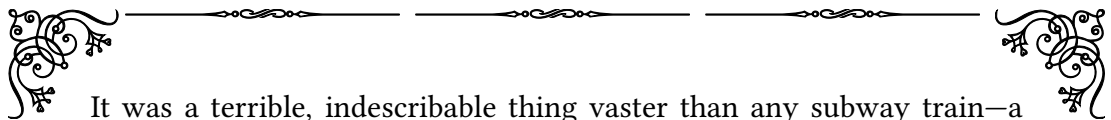
$$\begin{array}{c}
\frac{\sigma \vdash v : T}{\Gamma \vdash \langle v : T \rangle_{\sigma}^l \dashv \Gamma} \text{ (T-CONST)} \qquad \frac{\Gamma_1 \vdash \langle t : T \rangle_{\sigma}^l \dashv \Gamma_2}{\Gamma_1 \vdash \langle \text{box } t : \Box T \rangle_{\sigma}^l \dashv \Gamma_2} \text{ (T-BOX)} \\
\\
\frac{\Gamma \vdash w : \langle T \rangle^m \quad \text{copy}(T) \quad \neg \text{readProhibited}(\Gamma, w)}{\Gamma \vdash \langle !w : T \rangle_{\sigma}^l \dashv \Gamma} \text{ (T-COPY)} \\
\\
\frac{\Gamma \vdash w : \langle T \rangle^m \quad \neg \text{writeProhibited}(\Gamma_1, w) \quad \Gamma_2 = \text{move}(\Gamma_1, w)}{\Gamma_1 \vdash \langle \#w : T \rangle_{\sigma}^l \dashv \Gamma_2} \text{ (T-MOVE)} \\
\\
\frac{\Gamma \vdash w : \langle T \rangle^m \quad \text{mut}(\Gamma, w) \quad \neg \text{writeProhibited}(\Gamma, w)}{\Gamma \vdash \langle \&\text{mut } w : \&\text{mut } w \rangle_{\sigma}^l \dashv \Gamma} \text{ (T-MUTBORROW)} \\
\\
\frac{\Gamma \vdash w : \langle T \rangle^m \quad \neg \text{readProhibited}(\Gamma, w)}{\Gamma \vdash \langle \&w : \&w \rangle_{\sigma}^l \dashv \Gamma} \text{ (T-IMMBORROW)} \\
\\
\frac{\Gamma_1 \vdash \langle t_1 : T_1 \rangle_{\sigma}^l \dashv \Gamma_2 \quad \dots \quad \Gamma_n \vdash \langle t_n : T_n \rangle_{\sigma}^l \dashv \Gamma_{n+1}}{\Gamma_1 \vdash \langle \bar{t} : \bar{T}_n \rangle_{\sigma}^l \dashv \Gamma_{n+1}} \text{ (T-SEQ)} \\
\\
\frac{\Gamma_1 \vdash \langle t : T \rangle_{\sigma}^m \dashv \Gamma_2 \quad \Gamma_2 \vdash T \geq l \quad \Gamma_3 = \text{drop}(\Gamma_2, m)}{\Gamma_1 \vdash \langle \{t\} : T \rangle_{\sigma}^l \dashv \Gamma_3} \text{ (T-BLOCK)} \\
\\
\frac{\Gamma_1 \vdash \langle t_1 : T \rangle_{\sigma}^l \dashv \Gamma_2 \quad \Gamma_3 = \Gamma_2[x \mapsto \langle T \rangle^l] \quad \Gamma_3 \vdash t_2 : T' \dashv \Gamma_3}{\Gamma_1 \vdash \langle \text{let mut } x = t_1; t_2 : T' \rangle_{\sigma}^l \dashv \Gamma_3} \text{ (T-DECLARE)} \\
\\
\frac{\Gamma_1 \vdash w : \langle \tilde{T}_1 \rangle^m \quad \Gamma_1 \vdash \langle t : T_2 \rangle_{\sigma}^l \dashv \Gamma_2 \quad \Gamma_2 \vdash \tilde{T}_1 \approx T_2 \quad \Gamma_2 \vdash T_2 \geq m \quad \Gamma_3 = \text{write}^0(\Gamma_2, w, T_2) \quad \neg \text{writeProhibited}(\Gamma_3, w)}{\Gamma_1 \vdash \langle w = t : \epsilon \rangle_{\sigma}^l \dashv \Gamma_3} \text{ (T-ASSIGN)}
\end{array}$$

Figure 3.12: Typing Rules for FR

Chapter 4

Capturing A Shape-Shifter: The Semantic Process

A Formal Foundation for Strategic Rewriting




It was a terrible, indescribable thing vaster than any subway train—a shapeless congeries of protoplasmic bubbles, faintly self-luminous, and with myriads of temporary eyes forming and unforming as pustules of greenish light all over the tunnel-filling front that bore down upon us ... And at last we remembered that the daemoniac *shoggoths* — given life, thought, and plastic organ patterns solely by the Old Ones, and having no language save that which the dot-groups expressed — had likewise no voice save the imitated accents of their bygone masters.

— H. P. Lovecraft “From the Mountains of Madness”



Prologue

 HOGGOTH, as illustrated in the quotation, is a Lovecraftian *shape-shifting monster*, making the sound “Tekeli-li, Tekeli-li” which can no longer be understood by anyone. In this chapter, we thoroughly discuss a foundational study of a domain specific programming language — a strategic rewriting language for *syntactic transformations*, which previously lacked a formal treatment. By making

a metaphorical connection, we named the study Shoggoth and published it under the title *Shoggoth: A Formal Foundation for Strategic Rewriting*.

In chapter 1, we briefly discussed that conceptually it is intriguing to explore the relationship between syntax and semantics within the context of strategic rewriting. Since term rewriting, as a technique of syntactic transformations, encodes the semantics of some programs, and compositions of each individual term rewriting step form strategies, of which the straightforward syntax is given by a strategic rewriting language, whose semantics featuring non-termination and non-deterministic executions is complicated and worth studying to allow us to formally understand and reason about the composition of these rewrites.

In this chapter, we utilise three formal semantics models of programming languages, namely, denotational semantics, big-step operational semantics, and axiomatic semantics, to analyse a core calculus of a set of strategic rewriting languages, and discuss how these models relate to each other.

4.1 Introduction

Strategic rewriting allows programmers to compose rewrite rules and control their application. Dedicated strategy languages, such as Stratego (Visser et al., 1998; Visser, 2001) and more recently Elevate (Hagedorn et al., 2023, 2020), provide combinators for composing rewrite rules into larger strategies, as well as traversals to describe the location at which rewrite strategies are applied.

Strategic rewriting has many important practical applications. For instance, Stratego is used to specify the semantics of programming languages by writing interpreters with rewrite strategies in the Spoofox language workbench (Wachsmuth et al., 2014). Elevate is used to describe compiler optimisations for generating fast code achieving competitive performance to the state-of-the art machine learning compiler TVM (Hagedorn et al., 2020). Strategic rewriting is also used in domains ranging from generic programming (Lämmel and Visser, 2002) to tactic languages in proof assistants (Sozeau, 2014).

Compositions of rewrites easily become complex. For example, Hagedorn et al. (2020) report that for performing their compiler optimisations up to 60,000 rewrite steps are required. To orchestrate such long rewrite sequences, strategy languages provide various combinators for composing strategies together and traversals for applying strategies to different sub-expressions within the given abstract syntax tree.

Together with support for recursion, these combinators and traversals are capable of modelling the complex rewrite sequences required in practical applications.

This capability comes at the cost of semantic complexity, as strategies can be non-deterministic, they may give an error which triggers backtracking, and they may diverge due to the presence of general recursion. Such a combination of features introduces a lot of semantic subtleties, which make it easy to define not well-behaved strategies by mistake. For example, a strategy that does not terminate as it repeatedly tries to apply a rewrite. Similarly, it is easy to compose incompatible rewrites that will fail for every possible input expression. Finally, even if a rewrite strategy successfully terminates, it may not do what it was supposed to do by rewriting the input expression into an undesired form.

The goal of this chapter is to provide a rigorous treatment of strategic rewriting, that we believe lacks so far. Considering that strategic rewriting has various application domains but has problematic behaviours, a rigorous formal understanding of strategic rewriting is required to model and analyse its semantic subtleties as well as reason about the execution of strategies. Therefore, we present Shoggoth: a formal foundation for reasoning about strategic rewriting.

We start with introducing the formal syntax of *System S*, a formal core strategy language originally introduced by Visser and Benaissa (1998). Some example strategies are sketched to give the gist of strategic rewriting as well. We then give a comprehensive semantic accounting of strategic rewriting languages. We define a *denotational semantics* for System S, which originally had been given a *big-step operational semantics*. Our denotational semantics accounts for non-determinism and errors, and, unlike previous work, also explicitly models divergence. In addition, we formalise an extended big-step operational semantics which accounts for diverging executions, and formally prove the equivalence of our two models via soundness and computational adequacy theorems. All of our results have been mechanised in Isabelle/HOL (Nipkow et al., 2002).

To facilitate formal reasoning about rewriting strategies, we define a *weakest precondition calculus* that for a given postcondition computes the weakest precondition that must hold in order for the given strategy to execute successfully and satisfy the postcondition. Because traversals allow us to apply strategies to sub-expressions of the input expression, we must know not just which rewrite rules are to be applied, but also *where* in the input expression they are to be applied, in order to determine the weakest precondition. To accomplish this, our weakest precondition calculus is

location-based: weakest preconditions are not just based on the given strategy and desired postcondition, but also depend on the location at which the strategy is to be applied in the input expression. We have mechanised the definition of the weakest precondition calculus in Isabelle/HOL and formally proven its soundness with respect to the denotational semantics.

Finally, we show how to use the weakest precondition calculus to reason about rewrite strategies by applying it to various examples, including termination, that a strategy is well-composed, and that a rewrite strategy satisfies a particular postcondition after its execution. One of our examples is a strategy for $\beta\eta$ -normalisation taken from the Elevate project by Hagedorn et al. (2020), demonstrating the applicability of our work to practical scenarios.

In summary, we make the following contributions:

- We design, formalise and mechanise using Isabelle/HOL the semantics of System S, including both denotational and operational models with a full accounting of nondeterminism, errors, and divergence. We prove these two semantics equivalent (Section 4.3).
- We design, formalise and mechanise using Isabelle/HOL a location-based weakest precondition calculus for System S. We prove its soundness with respect to the denotational semantics (Section 4.4).
- We demonstrate how to use the weakest precondition calculus to prove practical useful properties of strategic rewriting (Section 4.5):
 - that a strategy terminates, i.e., that it does not diverge;
 - that a strategy is well-composed, i.e., that there exist input expressions for which the strategy execution will succeed;
 - that a desired property is satisfied after execution of the strategy.

Before stepping into the formalisation of System S, in the next section we present the syntax of System S as well as some example strategies to facilitate the understanding of strategic rewriting.

4.2 The Syntax of System S

System S (Visser and Benaissa, 1998) is a core calculus providing basic constructs of strategic rewriting, including atomic strategies (rewrite rules) and operators composing strategies and performing expression traversals in an abstract syntax tree (AST). A successful execution of a strategy transforms an expression into some other expression while preserving its semantics. The expressions being rewritten can either be *Leaf*s or *nodes*, in general, taking the form of:

$$\text{Expression}(\mathbb{E}) \quad e ::= \text{Leaf} \mid \overset{n}{e \wedge e}$$

Figure 4.1 presents the syntax of strategies in System S. We use \mathbb{S} to denote the set of all strategies. Variables, atomic strategies, SKIP and ABORT are *basic strategies*. Basic strategies are not decomposable. An atomic strategy is simply a rewrite rule. For instance, the commutativity of addition add_{com} and commutativity of multiplication $mult_{com}$ are atomic strategies:

$$\begin{array}{ll} add_{com} : a + b \rightsquigarrow b + a & \text{Commutativity of addition} \\ mult_{com} : a * b \rightsquigarrow b * a & \text{Commutativity of multiplication} \end{array}$$

SKIP can always be executed successfully while executing ABORT would always cause failure. To compose strategies, one can make use of *combinators* including sequential composition ($;$), left choice ($<+$) and nondeterministic choice ($<+>$). Sequential composition instructs to execute two strategies one after the other. Left choice prefers executing the strategy at the left hand side of the combinator over the strategy at the right hand side of the operator while nondeterministic choice decides to execute one of the given two strategy nondeterministically. In addition, *one*, *some* and *all* are *traversals* that navigate within the AST. Intuitively, $one(s)$ applies s to one immediate sub-expression of an input expression, $some(s)$ applies s to as many immediate sub-expressions of an input expression as possible and $all(s)$ applies s to all immediate sub-expressions of an input expression. Lastly, System S provides a *fixed-point operator* to model recursion.

Comparison of the expressiveness to the original System S One difference between our formalism and the original System S is that we abstract away the term building details for atomic strategies, instead modelling atomic strategies as partial functions. We believe that applying this abstraction does not limit the expressiveness

$$\begin{aligned}
\text{Strategy}(\mathbb{S}) \quad s ::= & \text{atomic} \mid X \mid \text{SKIP} \mid \text{ABORT} \\
& \mid s ; s \mid s <+ s \mid s <+> s \\
& \mid \text{one}(s) \mid \text{some}(s) \mid \text{all}(s) \\
& \mid \mu X.s
\end{aligned}$$

Figure 4.1: The Syntax of System S

of our system. In fact, the purpose of such design is to allow the flexibility of the term languages, not only limited to the original System S, but also capturing other strategic rewriting languages that use term constructs that are different from System S. Moreover, this design enables us to focus on reasoning about properties of compositions of rewriting strategies that hold independent of the term building behaviour.

Composing strategies We can compose strategies together with these combinators, traversals and the fixed-point operator to define more strategies. For example, we define a strategy $\text{try}(s)$ using left choice and SKIP which attempts to apply a strategy s to an input expression. If an error occurs, then it will leave the input expression unchanged by executing the strategy SKIP:

$$\text{try}(s) := s <+ \text{SKIP}$$

With the fixed-point operator and sequential composition, we can then define a strategy $\text{repeat}(s)$ which keeps applying a strategy s to an input expression until its no longer applicable:

$$\text{repeat}(s) := \mu X. \text{try}(s ; X)$$

With the fixed-point operator, the traversal $\text{one}(s)$ and left choice, we can define top-down and bottom-up traversals in an AST:

$$\text{topDown}(s) := \mu X. (s <+ \text{one}(X)) \quad \text{bottomUp}(s) := \mu X. (\text{one}(X) <+ s)$$

We can further compose $\text{repeat}(s)$ and $\text{topDown}(s)$ to define a strategy $\text{normalise}(s)$, which keeps applying a strategy s to all sub-expressions of an input expression until it is no longer applicable:

$$\text{normalise}(s) := \text{repeat}(\text{topDown}(s))$$

The normalise strategy is very commonly used for expressing program transformations. Given beta and eta reductions for λ -expressions, we can use the normalisation

strategy $normalise(beta <+ eta)$ for normalising an input λ -expression into its $\beta\eta$ -normal form.

As previously mentioned, the composition of strategies can be invalid and the executions of strategies are not always successful. For instance, the strategy $mult_{com} ; add_{com}$ is not well composed since it cannot be successfully executed on any input expression. $repeat(SKIP)$ is a strategy that cannot terminate. Although $normalise(beta <+ eta)$ can certainly be successfully executed on some input expressions, on other inputs it may not terminate. It is important to know that when it terminates, it will indeed leave the expression in $\beta\eta$ -normal form.

To reason about the successful and unsuccessful executions of strategies, we design the *location-based weakest precondition calculus* which is discussed in section 4.4. With this calculus, we are able to detect *bad* strategies that do not have successful executions, like $mult_{com} ; add_{com}$ and $repeat(SKIP)$, by concluding that there is no input expression that can be successfully rewritten by such strategies into a desired form. Also, for a *good* strategy that has successful executions, we are able to distinguish inputs that indeed lead to successful executions of the strategy and inputs that lead to erroneous or diverging executions. Such reasoning power is demonstrated in section 4.5.

To design the location-based weakest precondition calculus, we need to understand the behaviours of executing these strategies in System S. Therefore, before introducing the calculus and its reasoning power, we firstly study the formal semantics of System S.

4.3 The Semantics of System S

For given collections of expressions \mathbb{E} , System S defines nondeterministic executions for given strategies that can result in expressions or errors. We extend the original System S by allowing divergence as a possible result of executing a strategy. Thus, applying a strategy to an expression can result in expressions, an error or divergence.

4.3.1 The Plotkin Powerdomain

We provide a denotational semantics of System S as an instance of Plotkin's powerdomain construction (Plotkin, 1976), which allows us to assign least fixed points as the semantics of the recursion construct. An ω -complete partial order (ω -cpo) is

a partially ordered set (X, \leq) in which each ω -chain $(x_1 \leq x_2 \leq x_3 \leq \dots)$ has a least upper bound. A function $f : X \rightarrow X$ on such a set is *continuous* if for each ω -chain $x_1 \leq x_2 \leq x_3 \leq \dots$ with least upper bound x , one has that $f(x)$ is the least upper bound of the set $\{f(x_1), f(x_2), f(x_3), \dots\}$. A continuous function is certainly *monotone*, in the sense that $x_1 \leq x_2$ implies $f(x_1) \leq f(x_2)$ – this follows by considering the ω -chain $x_1 \leq x_2 \leq x_2 \leq x_2 \leq \dots$, and its least upper bound x_2 . Now Kleene's fixed-point theorem says that each continuous function f on an ω -cpo with a least element has a least fixed point.

Consider a nondeterministic, possibly diverging, algorithm that transforms values into values. If \mathcal{V} is the set of values, this algorithm can be modelled as a function $f : \mathcal{V} \rightarrow \mathcal{P}_{-\emptyset}(\mathcal{V}_\perp)$, where $\mathcal{P}_{-\emptyset}(X)$ is the set of non-empty subsets of X , the non-empty-powerset, and $\mathcal{V}_\perp := \mathcal{V} \uplus \{\perp\}$ is the set in which we embed \mathcal{V} together with a new element \perp . The newly added element \perp represents the outcome where the algorithm diverges. We equip the set \mathcal{V}_\perp with a partial order by defining:

$$x \leq y \iff x = \perp \vee x = y.$$

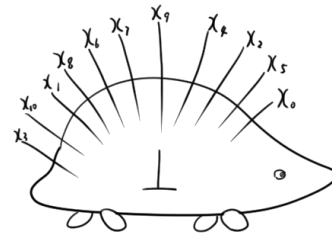
This fits with the intuition that \perp represents a computation that has not yet terminated, and $x \leq y$ holds when y is a later stage of the computation x .

Terminated computations are identified by the values they compute. We compare sets of values using the Egli-Milner ordering:

$$A \leq B \iff (\forall x \in A. \exists y \in B. x \leq y) \wedge (\forall y \in B. \exists x \in A. x \leq y)$$

Lifting a partial order from elements to sets in this fashion always yields a preorder. For a *flat domain* \mathcal{V}_\perp , \leq is a partial order on $\mathcal{P}_{-\emptyset}(\mathcal{V}_\perp)$. It is characterised by:

$$A \leq B \iff A = B \vee ((\perp \in A) \wedge A \setminus \{\perp\} \subseteq B)$$



(Porcupine ordering)

The resulting poset $\mathcal{P}_{-\emptyset}(\mathcal{V}_\perp)$ is an ω -cpo. Each ω -chain either enters a spine of the porcupine, and thus contains a largest element which is its least upper bound, or \perp is a member of all elements in the chain, so that its least upper bound is simply the union of all sets in the chain.

Aside on the powerdomain construction and the Egli-Milner ordering. To give some further insight into the powerdomain construction and the Egli-Milner

ordering, recall the following well-known characterisation. Hennessy and Plotkin (1979, Remark after Lemma 3.5) show that Plotkin's (1976) powerdomain construction extends to all ω -complete partial orders (ω -cpo) by sending each ω -cpo to the *free semi-lattice* over it. In detail, given an ω -cpo X , we define a *free semi-lattice over X* as an ω -cpo DX , together with a Scott-continuous function $\eta : X \rightarrow DX$ and a Scott-continuous binary operation: $\vee : (DX)^2 \rightarrow DX$ that is associative, commutative, and idempotent. A free semi-lattice always exists, but its explicit description may be complicated. Hennessy and Plotkin show that, when ω -cpo is ω -algebraic, we can construct the free semi-lattice explicitly by taking $DX := \mathcal{P}_{\perp} X$ to be the powerdomain construction with the Egli-Milner ordering, $\eta(x) := \{x\}$ as the embedding of X into this semi-lattice, and sub-set union as the binary operation. So in a specific and technical sense, the powerdomain DX is the simplest extension of the ω -cpo X with an associative, idempotent and commutative binary operator.

(end of aside)

In our mechanised Isabelle/HOL formalisation, we opt to use posets that are complete with respect to all chains, not merely countable or directed ones, without maintaining continuity as an assumption. The stronger assumption on posets allows us to weaken the assumption on functions: we only require monotonicity to ensure existence of fixed points. This choice was made purely for ease of formalisation, as Isabelle/HOL already includes a library for chain-complete partial orders. While this means that our domain may contain monotone functions that do not correspond to any expressible strategy, and that Hennessy and Plotkin's characterisation does not directly apply, our meta-theoretic results below show how to relate our semantics to the operational semantics, and our reasoning examples show that this semantics suffices to reason about practically interesting examples. We conjecture that our results will easily carry over to a semantics defined with ω -cpo.

4.3.2 Formalised Denotational Semantics

We now present and discuss the denotational semantics for System S, capturing successful and erroneous executions of strategies as well as nondeterminism, divergence and recursion. A strategy is a nondeterministic algorithm/function that rewrites expressions into expressions. This nondeterministic algorithm can sometimes yield an error *err* instead of an expression, and it might fail to terminate. In the latter case, we say that it yields the value *div*. Formally, we instantiate Plotkin's powerdomain

construction from the previous section by setting $\mathcal{V} := \mathbb{E} \cup \{err\}$ and $\perp := div$, noting it is a flat domain. We denote the resulting powerdomain by:

$$\begin{aligned} \mathcal{D}_p &:= \mathcal{P}_{-\emptyset}(\mathbb{E} \cup \{err\} \cup \{div\}) \quad \text{ordered by:} \\ A \leq B &\iff A = B \vee ((div \in A) \wedge A \setminus \{div\} \subseteq B) \end{aligned}$$

We define the denotational semantics of System S over the point-wise lifting of the powerdomain:

$$\mathcal{D} = \mathbb{E} \rightarrow \mathcal{D}_p$$

To define the denotational semantics of strategies in a concise manner, we provide semantic combinators and traversals that encapsulate the meaning of syntactic combinators and traversals.

Figure 4.2 illustrates the definitions of the combinators. The definition of sequential composition $s ;_s t$ is straightforward, indicating that the execution of the strategy t depends on the result of applying s to the input expression e . If applying s to e results in an error or divergence, the sequential composition will produce an error and divergence, respectively. Otherwise, the result of the sequential composition $s ;_s t$ is produced by applying t to the expression obtained by the execution of s . The definition of left choice $s <+_s t$ prioritises the execution of the strategy s over t . The strategy t will only be executed if the execution of s produces an error. Our treatment of nondeterminism is *demonic* with respect to divergence while *angelic* with respect to errors. If either the execution of s or t diverges, then the nondeterministic choice $s <+>_s t$ diverges as well. The nondeterministic choice will only result in an error if both executions of s and t result in an error. When both s and t give cause for a successful execution, the choice is nondeterministic.

These combinators are sufficient for composing strategies applied to the root of an AST. System S also provide traversals *one*, *some* and *all* to apply strategies to sub-expressions. Their semantics are shown in figure 4.3. The traversal $one_s(s)(e)$ nondeterministically chooses one immediate sub-expression of e and applies strategy s to it. The treatment of nondeterminism here is again demonic with respect to divergence and angelic with respect to errors. If applying s to one of the sub-expressions results in divergence, $one_s(s)$ will diverge. An error will only occur when e has no sub-expression or applying s to all sub-expressions of e results in error. The traversal $some_s(s)(e)$ applies s to as many immediate sub-expressions of e as possible. Its divergence and erroneous situations are the same as one_s . The successful execution of

$$(\cdot;_s) : \mathfrak{D} \rightarrow \mathfrak{D} \rightarrow \mathfrak{D}$$

$$(s;_s t)(e) = \bigcup \{t(e') \mid e' \in s(e) \cap \mathbb{E}\} \cup \{r \mid r \in s(e) \cap \{div, err\}\}$$

(Sequential composition)

$$(<+_s) : \mathfrak{D} \rightarrow \mathfrak{D} \rightarrow \mathfrak{D}$$

$$(s <+_s t)(e) = (s(e) \setminus \{err\}) \cup \{e' \mid e' \in t(e) \wedge err \in s(e)\}$$

(Left choice)

$$(<+>_s) : \mathfrak{D} \rightarrow \mathfrak{D} \rightarrow \mathfrak{D}$$

$$(s <+>_s t)(e) = \{e' \mid e' \in s(e) \cap \mathbb{E}\} \cup \{div \mid div \in s(e)\}$$

$$\cup \{e' \mid e' \in t(e) \cap \mathbb{E}\} \cup \{div \mid div \in t(e)\} \cup \{err \mid err \in s(e) \cap t(e)\}$$

(Nondeterministic choice)

Figure 4.2: Semantic Combinators of System S

$all_s(s)$ on an input expression e requires successful application of s to all immediate sub-expressions of e or e being a *Leaf*. If applying s to one sub-expression leads to an error or divergence, $all_s(s)(e)$ yields *err* or *div*, respectively. For simplicity of the presentation and illustration, we have restricted ourselves to binary trees in this study. However, the traversals can easily be generalised to ASTs with wider branching.

With the semantic combinators and semantic traversals introduced, we provide the denotational semantics for System S shown in figure 4.4. The semantics of a strategy is modelled as a function that takes in a *semantic environment* ξ , which is a function mapping variables to elements of \mathfrak{D} .

The semantics of a variable consists of looking up the variable in a given semantic environment. We model an atomic strategy as a partial function, which can successfully rewrite an input expression into an output expression when it is defined for the input expression. When an atomic strategy is not defined for an input expression, applying it to the input expression will result in an error. SKIP is a strategy that always rewrites an input expression to itself while ABORT is a strategy that always produces an *err*. The denotational semantics of combinators and traversals are straightforwardly defined with the semantic combinators and traversals. Lastly, the

$$(one_s) : \mathfrak{D} \rightarrow \mathfrak{D}$$

$$\begin{aligned} one_s(s)(e) = & \{ \overline{e'_1 e'_2}^n \mid e = \overline{e_1 e_2}^n \wedge e'_1 \in s(e_1) \cap \mathbb{E} \} \\ & \cup \{ \overline{e_1 e'_2}^n \mid e = \overline{e_1 e_2}^n \wedge e'_2 \in s(e_2) \cap \mathbb{E} \} \\ & \cup \{ div \mid e = \overline{e_1 e_2}^n \wedge div \in s(e_1) \cup s(e_2) \} \\ & \cup \{ err \mid e = Leaf \vee (e = \overline{e_1 e_2}^n \wedge err \in s(e_1) \cap s(e_2)) \} \end{aligned}$$

(One)

$$(some_s) : \mathfrak{D} \rightarrow \mathfrak{D}$$

$$\begin{aligned} some_s(s)(e) = & \{ \overline{e'_1 e'_2}^n \mid e = \overline{e_1 e_2}^n \wedge e'_1 \in s(e_1) \cap \mathbb{E} \wedge e'_2 \in s(e_2) \cap \mathbb{E} \} \\ & \cup \{ \overline{e'_1 e_2}^n \mid e = \overline{e_1 e_2}^n \wedge e'_1 \in s(e_1) \cap \mathbb{E} \wedge err \in s(e_2) \} \\ & \cup \{ \overline{e_1 e'_2}^n \mid e = \overline{e_1 e_2}^n \wedge e_2 \in s(e_2) \cap \mathbb{E} \wedge err \in s(e_1) \} \\ & \cup \{ div \mid e = \overline{e_1 e_2}^n \wedge div \in s(e_1) \cup s(e_2) \} \\ & \cup \{ err \mid e = Leaf \vee (e = \overline{e_1 e_2}^n \wedge err \in s(e_1) \cap s(e_2)) \} \end{aligned}$$

(Some)

$$(all_s) : \mathfrak{D} \rightarrow \mathfrak{D}$$

$$\begin{aligned} all_s(s)(e) = & \{ Leaf \mid e = Leaf \} \\ & \cup \{ \overline{e'_1 e'_2}^n \mid e = \overline{e_1 e_2}^n \wedge e'_1 \in s(e_1) \cap \mathbb{E} \wedge e'_2 \in s(e_2) \cap \mathbb{E} \} \\ & \cup \{ div \mid e = \overline{e_1 e_2}^n \wedge div \in s(e_1) \cup s(e_2) \} \\ & \cup \{ err \mid e = \overline{e_1 e_2}^n \wedge err \in s(e_1) \cup s(e_2) \} \end{aligned}$$

(All)

Figure 4.3: Semantic Traversals of System S

Variable(\mathbb{V})	$X\ Y\ Z\ \dots$
Semantic Environment(Γ_S)	$\xi : \mathbb{V} \rightarrow \mathfrak{D}$
$\llbracket \$ \rrbracket : \Gamma_S \rightarrow \mathfrak{D}$	
$\llbracket X \rrbracket \xi = \xi X$	
$\llbracket \text{atomic} \rrbracket \xi = \lambda e. \{ \text{atomic}(e) \mid \text{atomic}(e) \textbf{def} \} \cup \{ \text{err} \mid \text{atomic}(e) \textbf{undef} \}$	
$\llbracket \text{SKIP} \rrbracket \xi = \lambda e. \{ e \}$	
$\llbracket \text{ABORT} \rrbracket \xi = \lambda e. \{ \text{err} \}$	
$\llbracket s ; t \rrbracket \xi = \llbracket s \rrbracket \xi ;_s \llbracket t \rrbracket \xi$	(Sequential composition)
$\llbracket s <+ t \rrbracket \xi = \llbracket s \rrbracket \xi <+_s \llbracket t \rrbracket \xi$	(Left choice)
$\llbracket s <+> t \rrbracket \xi = \llbracket s \rrbracket \xi <+>_s \llbracket t \rrbracket \xi$	(Nondeterministic choice)
$\llbracket \text{one}(s) \rrbracket \xi = \text{one}_s(\llbracket s \rrbracket \xi)$	(One)
$\llbracket \text{some}(s) \rrbracket \xi = \text{some}_s(\llbracket s \rrbracket \xi)$	(Some)
$\llbracket \text{all}(s) \rrbracket \xi = \text{all}_s(\llbracket s \rrbracket \xi)$	(All)
$\llbracket \mu X. s \rrbracket \xi = \mu \mathcal{X}. \llbracket s \rrbracket (\xi[X \mapsto \mathcal{X}])$	(Fixed point)

Figure 4.4: Denotational Semantics of System S

semantics of the fixed-point operator is the least fixed point in our domain, where we extend the semantic environment with a mapping from the syntactic fixed-point variable to the fixed point in our domain. We denote this environment extension with the syntax $\xi[X \mapsto d]$.

The denotational semantics is monotone Given two environments ξ_1 and ξ_2 , if the values obtained from looking up the variables in the environments satisfy the ordering $\xi_1(X) \leq \xi_2(X)$ for any variable X , the values obtained from evaluation of a strategy s with these environments should also satisfy the ordering $\llbracket s \rrbracket \xi_1 \leq \llbracket s \rrbracket \xi_2$. Formally, we present the monotonicity theorem 4.3.1:

Theorem 4.3.1 (Semantics monotonicity theorem). *For given environments ξ_1 and ξ_2 , and strategy s we have:*

$$\frac{\forall X. \xi_1(X) \leq \xi_2(X)}{\llbracket s \rrbracket \xi_1 \leq \llbracket s \rrbracket \xi_2}$$

We prove this theorem in Isabelle/HOL by structural induction on the strategy s .

$$\begin{array}{c}
\frac{}{e \xrightarrow{\text{SKIP}} e} \text{ (SKIP)} \qquad \frac{}{e \xrightarrow{\text{ABORT}} \text{err}} \text{ (ABORT)} \qquad \frac{}{e \xrightarrow{\text{atomic}} \text{atomic}(e)} \text{ (ATOMIC)} \\
\\
\frac{e \xrightarrow{s_1} e_1 \quad e_1 \xrightarrow{s_2} e_2}{e \xrightarrow{s_1 ; s_2} e_2} \text{ (SEQCOMP)} \qquad \frac{e \xrightarrow{s_1} \text{err}}{e \xrightarrow{s_1 ; s_2} \text{err}} \text{ (SEQCOMPERR(1))} \\
\\
\frac{e \xrightarrow{s_1} e_1 \quad e_1 \xrightarrow{s_2} \text{err}}{e \xrightarrow{s_1 ; s_2} \text{err}} \text{ (SEQCOMPERR(2))} \qquad \frac{e \xrightarrow{s_1} e_1}{e \xrightarrow{s_1 < + s_2} e_1} \text{ (LCHOICE (L))} \\
\\
\frac{e \xrightarrow{s_1} \text{err} \quad e \xrightarrow{s_2} e_2}{e \xrightarrow{s_1 < + s_2} e_2} \text{ (LCHOICE (R))} \qquad \frac{e \xrightarrow{s_1} \text{err} \quad e \xrightarrow{s_2} \text{err}}{e \xrightarrow{s_1 < + s_2} \text{err}} \text{ (LCHOICEERR)} \\
\\
\frac{e \xrightarrow{s_1} e_1}{e \xrightarrow{s_1 < + s_2} e_1} \text{ (CHOICE(L))} \qquad \frac{e \xrightarrow{s_2} e_2}{e \xrightarrow{s_1 < + s_2} e_2} \text{ (CHOICE(R))} \qquad \frac{e \xrightarrow{s_1} \text{err} \quad e \xrightarrow{s_2} \text{err}}{e \xrightarrow{s_1 < + s_2} \text{err}} \text{ (CHOICEERR)} \\
\\
\frac{e \xrightarrow{s[X := \mu X.s]} e_1}{e \xrightarrow{\mu X.s} e_1} \text{ (FIXEDPOINT)} \qquad \frac{e \xrightarrow{s[X := \mu X.s]} \text{err}}{e \xrightarrow{\mu X.s} \text{err}} \text{ (FIXEDPOINTERR)}
\end{array}$$

Figure 4.5: Big-step operational semantics of non-diverging cases for basic strategies, combinators, and the fixed-point operator

4.3.3 Formalised Big-Step Operational Semantics

In this section, we present the formalised big-step operational semantics of System S, with our extension allowing for divergent strategies. Figure 4.5 depicts the big-step operational semantics for the non-diverging cases of System S. These cases are essentially the same as those of Visser and Benaissa (1998), albeit with the aforementioned simplification to binary trees applied.¹ The semantic rules are given in a straightforward way.

On top of these rules for terminating cases, we define the semantics for divergence as the *coinductive* judgement Leroy and Grall (2009) satisfying the rules shown in figure 4.7. Here we use $e \xrightarrow{\infty}^s$ to indicate that the evaluation of an expression e by a strategy s leads to divergence.

¹Visser and Benaissa (1998) denote error by \uparrow .

$$\begin{array}{c}
\frac{}{Leaf \xrightarrow{one(s)} err} \text{ (ONE(Id))} \quad \frac{}{Leaf \xrightarrow{some(s)} err} \text{ (SOME(Id))} \quad \frac{}{Leaf \xrightarrow{all(s)} Leaf} \text{ (ALL(Id))} \\
\\
\frac{e_1 \xrightarrow{s} e'_1}{\frac{n}{e_1 \ e_2} \xrightarrow{one(s)} \frac{n}{e'_1 \ e_2}} \text{ (ONE(L))} \quad \frac{e_2 \xrightarrow{s} e'_2}{\frac{n}{e_1 \ e_2} \xrightarrow{one(s)} \frac{n}{e_1 \ e'_2}} \text{ (ONE(R))} \\
\\
\frac{e_1 \xrightarrow{s} err \quad e_2 \xrightarrow{s} err}{\frac{n}{e_1 \ e_2} \xrightarrow{one(s)} err} \text{ (ONEERR)} \\
\\
\frac{e_1 \xrightarrow{s} e'_1 \quad e_2 \xrightarrow{s} err}{\frac{n}{e_1 \ e_2} \xrightarrow{some(s)} \frac{n}{e'_1 \ e_2}} \text{ (SOME(L))} \quad \frac{e_1 \xrightarrow{s} err \quad e_2 \xrightarrow{s} e'_2}{\frac{n}{e_1 \ e_2} \xrightarrow{some(s)} \frac{n}{e_1 \ e'_2}} \text{ (SOME(R))} \\
\\
\frac{e_1 \xrightarrow{s} e'_1 \quad e_2 \xrightarrow{s} e'_2}{\frac{n}{e_1 \ e_2} \xrightarrow{some(s)} \frac{n}{e'_1 \ e'_2}} \text{ (SOME)} \quad \frac{e_1 \xrightarrow{s} err \quad e_2 \xrightarrow{s} err}{\frac{n}{e_1 \ e_2} \xrightarrow{some(s)} err} \text{ (SOMEERR)} \\
\\
\frac{e_1 \xrightarrow{s} e'_1 \quad e_2 \xrightarrow{s} e'_2}{\frac{n}{e_1 \ e_2} \xrightarrow{all(s)} \frac{n}{e'_1 \ e'_2}} \text{ (ALL)} \quad \frac{e_1 \xrightarrow{s} err}{\frac{n}{e_1 \ e_2} \xrightarrow{all(s)} err} \text{ (ALLERR(L))} \\
\\
\frac{e_2 \xrightarrow{s} err}{\frac{n}{e_1 \ e_2} \xrightarrow{all(s)} err} \text{ (ALLERR(R))}
\end{array}$$

Figure 4.6: Big-step operational semantics of non-diverging cases for traversals

$$\begin{array}{c}
\frac{e \xrightarrow[\infty]{s_1}}{\quad} \text{(SEQCOMPDiv(1))} \quad \frac{e \xrightarrow{s_1} e_1 \quad e_1 \xrightarrow[\infty]{s_2}}{e \xrightarrow[\infty]{s_1 ; s_2}} \text{(SEQCOMPDiv(2))} \\
\\
\frac{e \xrightarrow[\infty]{s_1}}{\quad} \text{(LCHOICEDiv(1))} \quad \frac{e \xrightarrow{s_1} \text{err} \quad e \xrightarrow[\infty]{s_2}}{e \xrightarrow[\infty]{s_1 <+ s_2}} \text{(LCHOICEDiv(2))} \\
\\
\frac{e \xrightarrow[\infty]{s_1}}{\quad} \text{(CHOICEDiv(1))} \quad \frac{e \xrightarrow[\infty]{s_2}}{\quad} \text{(CHOICEDiv(2))} \\
\\
\frac{e_1 \xrightarrow[\infty]{s}}{\quad} \text{(ONEDiv(1))} \quad \frac{e_2 \xrightarrow[\infty]{s}}{\quad} \text{(ONEDiv(2))} \\
\\
\frac{n \quad \frac{e_1 \quad e_2}{\quad} \xrightarrow[\infty]{\text{one}(s)}}{\quad} \text{(SOMEDiv (1))} \quad \frac{n \quad \frac{e_2 \quad \quad}{\quad} \xrightarrow[\infty]{\text{one}(s)}}{\quad} \text{(SOMEDiv (2))} \\
\\
\frac{n \quad \frac{e_1 \quad e_2}{\quad} \xrightarrow[\infty]{\text{some}(s)}}{\quad} \text{(ALLDiv (1))} \quad \frac{n \quad \frac{e_2 \quad \quad}{\quad} \xrightarrow[\infty]{\text{some}(s)}}{\quad} \text{(ALLDiv (2))} \\
\\
\frac{n \quad \frac{e_1 \quad e_2}{\quad} \xrightarrow[\infty]{\text{all}(s)}}{\quad} \text{(ALLDiv (1))} \quad \frac{n \quad \frac{e_2 \quad \quad}{\quad} \xrightarrow[\infty]{\text{all}(s)}}{\quad} \text{(ALLDiv (2))} \\
\\
\frac{e \xrightarrow[\infty]{s[X:=\mu X.s]}}{\quad} \text{(FIXEDPOINTDiv)} \\
\frac{\mu X.s}{e \xrightarrow[\infty]{\quad}}
\end{array}$$

Figure 4.7: Big-step operational semantics of diverging cases

4.3.4 The Denotational Semantics is Equivalent to The Big-Step Operational Semantics

In section 4.3.2 and section 4.3.3, we have provided two styles of semantics for System S. It is essential to prove that these two semantics are equivalent, since we would like our formal semantics to provide unambiguous and unique interpretation of strategies in System S. In addition, with the equivalence of these two semantics established, we only need to refer to one of them to prove some properties of System S and they should also hold for the other semantics.

We reason about the equivalence between the denotational semantics and big-step operational semantics via computational soundness and computational adequacy theorems. More specifically, we have a pair of computational soundness and adequacy theorems to relate the non-diverging cases and a pair of computational soundness and adequacy theorems to relate the diverging cases.

Firstly, we show that if an expression e is evaluated to another expression or an error using the big-step operational semantics of a strategy s_\clubsuit , this result must also be in the set obtained by executing the denotational semantics of s_\clubsuit with the given expression e . Formally, this is described by our first computational soundness theorem 4.3.2. The subscript \clubsuit indicates that s_\clubsuit is a *closed* strategy: a strategy with no free variables, i.e. $fv(s_\clubsuit) = \emptyset$.

Theorem 4.3.2 (Computational soundness theorem one). *For a given closed strategy s_\clubsuit , and any environment ξ , we have for an arbitrary expression e and result r :*

$$\frac{e \xrightarrow{s_\clubsuit} r}{r \in \llbracket s_\clubsuit \rrbracket \xi e}$$

We prove this theorem by induction on the derivation of $e \xrightarrow{s_\clubsuit} r$ from the rules of figure 4.5. As the strategy s_\clubsuit is always closed, to instantiate our inductive hypothesis in the cases for the fixed-point operator, we make use of the following substitution lemma 4.3.3 to semantically relate the syntactic substitution of a closed strategy s_\clubsuit for a variable X in s with the strategy s under the environment where X maps to the semantics of s_\clubsuit .

Lemma 4.3.3 (Substitution lemma one).

$$\llbracket s[X := s_\clubsuit] \rrbracket \xi = \llbracket s \rrbracket \xi[X \mapsto (\llbracket s_\clubsuit \rrbracket \xi)]$$

This lemma can easily be generalised to allow s_\bullet to instead be an open strategy, so long as X is not free in s_\bullet , however our operational semantics only ever substitutes closed strategies, thus this generalisation is not necessary for proving our semantic equivalence theorems.

We now prove a computational adequacy theorem, the converse of the computational soundness theorem 4.3.2. It states that if a non-diverging result r is one of the results of the denotational semantics for a closed strategy s_\bullet with an input expression e , then the big-step operational semantics of s_\bullet with the input expression e will produce the same result.

Theorem 4.3.4 (Computational adequacy theorem one). *For an expression e , result r , and closed strategy s_\bullet we have:*

$$\frac{r \in \llbracket s_\bullet \rrbracket \xi e \quad r \neq \text{div}}{e \xrightarrow{s_\bullet} r}$$

To prove this theorem, we first generalise to open strategies. To do this, we define an approximation relation between a closed strategy and an element of our domain, and state an approximation lemma. Here, we employ, for a simultaneous substitution $\theta : \mathbb{V} \rightarrow \mathbb{S}_\bullet$, the notation $s[\theta]$ for the application of θ to all free variables in s .

Definition 4.3.1 (Approximation relation one). *Given a closed strategy s_\bullet and a function $d \in \mathcal{D}$, we say $s_\bullet \triangle d$ if and only if for any given input expression e , when r is a non-diverging result obtained by applying d to e , r will also be the result of evaluating the big-step operational semantics of s_\bullet with the input expression e .*

$$s_\bullet \triangle d \iff \forall e. r. r \in d(e) \cap (\mathbb{E} \cup \{\text{err}\}) \Rightarrow e \xrightarrow{s_\bullet} r$$

Lemma 4.3.5 (Approximation lemma one).

$$\frac{\forall y \in \text{fv}(s). \theta(y) \triangle \xi(y) \quad s_\bullet = s[\theta]}{s_\bullet \triangle \llbracket s \rrbracket \xi}$$

The proof of this lemma is by induction on the strategy s , and Scott induction is required for the fixed point cases. From the approximation lemma, we prove the computational adequacy theorem 4.3.4 by setting $s := s_\bullet$. As there are no free variables in s_\bullet , the approximation relation trivially implies our goal.

The computational soundness and adequacy theorems presented above state that the denotational semantics and big-step operational semantics are equivalent for the

non-diverging cases. Next, we present computational soundness and adequacy theorems for divergent strategies.

The computational soundness theorem for the diverging cases states that, if the evaluation of the big-step operational semantics of a closed strategy s_\bullet with an input expression e diverges, div must be in the resulting set obtained by executing the denotational semantics of s_\bullet with the given expression e .

Theorem 4.3.6 (Computational soundness theorem two).

$$\frac{e \xrightarrow[\infty]{s_\bullet}}{div \in \llbracket s_\bullet \rrbracket \xi e}$$

Just as with computational adequacy for non-diverging cases, we must first generalise to open strategies. We define the second approximation relation together with an approximation lemma to prove this soundness theorem.

Definition 4.3.2 (Approximation relation two). *Given a closed strategy s_\bullet and a function $d \in \mathcal{D}$, we say $s_\bullet \triangle_\infty d$ if and only if for any given input expression e , when evaluating the big-step operational semantics of s_\bullet with the input expression e diverges, div will be obtained by applying d to e , and we have the ordering $d(e) \leq \llbracket s_\bullet \rrbracket \xi e$.*

$$s_\bullet \triangle_\infty d \iff \forall e. (e \xrightarrow[\infty]{s_\bullet} \Rightarrow div \in d(e)) \wedge d(e) \leq \llbracket s_\bullet \rrbracket \xi e$$

Lemma 4.3.7 (Approximation lemma two).

$$\frac{\forall y \in fv(s). \theta(y) \triangle_\infty \xi(y) \quad s_\bullet = s[\theta]}{s_\bullet \triangle_\infty \llbracket s \rrbracket \xi}$$

The proof of this lemma is (again) by induction on the strategy s , where Scott induction is used for the fixed point cases. For the cases which involve terminating sub-steps, such as sequential composition or left choice, we make use of our soundness theorem for non-diverging cases, theorem 4.3.2. We utilise this approximation lemma 4.3.7 to prove the computational soundness theorem 4.3.6.

Lastly, we prove the computational adequacy theorem for the diverging cases, which is again the converse of the soundness theorem 4.3.6. It states that if div is in a result of executing the denotational semantics of a closed strategy s_\bullet with an input expression e , then evaluating the big-step operational semantics with the given expression e leads to divergence.

Theorem 4.3.8 (Computational adequacy theorem two).

$$\frac{div \in \llbracket s_\bullet \rrbracket \xi e}{e \xrightarrow[\infty]{s_\bullet}}$$

We prove this by *coinduction* over big-step operational semantics for diverging cases while making use of the computational adequacy theorem 4.3.4 for the non-diverging cases. Just as with our computational soundness proof for non-diverging cases, we work only with closed strategies s_\bullet , and rely on our substitution lemma 4.3.3 for the fixed point cases.

With these two pairs of computational soundness and adequacy theorems, we can conclude that the denotational semantics and big-step operational semantics are equivalent. Formally, we obtain:

Theorem 4.3.9 (Equivalence between semantics).

$$\llbracket s_\bullet \rrbracket \xi e = \{r \mid e \xrightarrow{s_\bullet} r\} \cup \{div \mid e \xrightarrow[\infty]{s_\bullet}\}$$

In this section, we have studied two styles of semantics of System S, namely a denotational semantics and a big-step operational semantics. To complete our semantic accounting, it may be worthwhile for us to study its small-step operational semantics in the future.

4.4 Location-Based Weakest Precondition Calculus

As we have seen, a strategy either successfully rewrites an expression into another expression, generates an error, or fails to terminate.

Naturally, we care mainly about the *successful* executions of a strategy. In particular, when it rewrites an input expression into another expression that satisfies a desired property. In order to formally understand successful and unsuccessful executions of strategies, we design and formalise a location-based weakest precondition calculus. Weakest preconditions were introduced by Dijkstra (1975), as an axiomatic semantics for his guarded command language. Different from other weakest precondition calculi, we introduce the notion of a *location* in an AST as a parameter in our calculus for reasoning about traversals, which is discussed in section 4.4.1. Before presenting the formal definition of the calculus, we recapitulate the definition of a weakest precondition.

Definition 4.4.1 (Weakest precondition). *Given a program S and a postcondition P , the weakest precondition $wp(S, P)$ is an assertion R_w such that for any precondition R :*

$$\{R\}S\{P\} \Leftrightarrow (R \Rightarrow R_w)$$

Here $\{R\}S\{P\}$ is a Hoare triple stating that S will successfully terminate in a state satisfying assertion P if the state before executing S satisfies R . Intuitively, the weakest precondition of S under P characterises all those states that lead to successful termination in a state of P when executing S . In Dijkstra's (1975) calculus, a function wp is defined which, given a program and an assertion as a postcondition, computes the weakest precondition inductively on the program structure. Bonsangue and Kok (1992) extend Dijkstra's calculus to assign weakest preconditions for a fixed-point operator by additionally including a *logic environment* as an input to the wp function, which associates a predicate transformer with each variable. As we also have a fixed-point operator for general recursion, we do the same in this formalisation.

When dealing with strategies, assertions take the form of sets of expressions, and a state is an expression we are rewriting. Thus, the weakest precondition is a set of input expressions for a strategy to be applied to, such that the result of the application of the strategy will lead to another expression. That means the strategy will neither yield an error nor diverge. Moreover, the weakest precondition has to guarantee that an expression of the postcondition is reached.

Definition 4.4.2 (Weakest must succeed precondition). *A weakest must succeed precondition takes the form $wp_{\zeta \Vdash s @ l}(P)$. This is the set of those expressions that, by applying strategy s at location l under the logic environment ζ , will be successfully transformed into expressions satisfying P .*

To calculate this set of input expressions constituting the weakest must succeed precondition, we also introduce the following auxiliary function. In fact, $wp_{\zeta \Vdash s @ l}(P)$ and $wp_{\zeta \Vdash s @ l}^\uparrow(P)$ will be defined by mutual induction.

Definition 4.4.3 (Weakest may error precondition). *A weakest may error precondition takes the form of $wp_{\zeta \Vdash s @ l}^\uparrow(P)$. This is the set of those expressions that, by applying strategy s at location l under the logic environment ζ , will be successfully transformed into expressions satisfying P , **or result in error**.*

4.4.1 Modelling Traversals

In definitions 4.4.2 and 4.4.3, we introduce the location for specifying the particular sub-expression to which the strategy s should be applied. This allows us to express that after applying a strategy s to the sub-expression *at the location* l of an input expression e , the input expression e should be transformed into an expression that satisfies the postcondition P . Consequently, the weakest precondition for traversals such as $one(s)$, $some(s)$, and $all(s)$ can be defined inductively in terms of the weakest precondition of s , just at different locations.

Kieburtz (2001) proposes an alternative approach, using modal logic for assertions about traversals. However, it is unclear how this technique could be used to define a complete calculus. We discuss this in section 4.6.

A *location* is essentially a path into the abstract syntax tree. Such a path consists of a sequence of positions, for our binary trees either left (ℓ) or right (r). Positions are prepended to a location with \triangleleft and appended with \triangleright . For instance, suppose we have an AST representing an arithmetic expression $1 + 3$, each sub-expression is located as:

$$\begin{array}{c} + (\epsilon) \\ \swarrow \quad \searrow \\ 1 (\epsilon \triangleright \ell) \quad 3 (\epsilon \triangleright r) \end{array}$$

With locations being introduced in the assertions, accompanied by the two helper functions *lookup* and *update* discussed in section 4.4.2, we can model the execution of a strategy at a given location in the input expression, which enables the assignments of weakest preconditions inductively for traversals just as with other operators.

4.4.2 The Calculus

We now introduce the location-based weakest precondition calculus for System S in its full formal detail. We first provide definitions of helper functions and essential notations for the formalisation.

To connect locations and expressions, we introduce two partial functions *lookup* and *update*, shown in figure 4.8. Given a location l and an expression e , the partial function *lookup* returns the sub-expression which is located at the location l in an expression e . The function is partial, as it is only defined when the location l actually exists in the expression e . The partial function *update* takes in a set $xs \in \mathfrak{D}_p$, and updates an expression e at the location l with each expression in xs , resulting in a set of

$lookup : \mathbb{L} \rightarrow \mathbb{E} \rightarrow \mathbb{E}$ (We write it as $\mathsf{h}_{l:\mathbb{L}}(e : \mathbb{E}) : (e' : \mathbb{E})$)

$lookup \in e = e$

$lookup(\ell \triangleleft l) \bigwedge_{e_1 e_2}^n = lookup\ l\ e_1$

$lookup(\ell' \triangleleft l) \bigwedge_{e_1 e_2}^n = lookup\ l\ e_2$

$update : \mathbb{L} \rightarrow \mathbb{E} \rightarrow \mathfrak{D}_p \rightarrow \mathfrak{D}_p$ (We write it as $(d : \mathfrak{D}_p) \sqsupseteq_{l:\mathbb{L}}(e : \mathbb{E}) : (d' : \mathfrak{D}_p)$)

$update \in e\ xs = xs$

$update(\ell \triangleleft l) \bigwedge_{e_1 e_2}^n xs = \{ \bigwedge_{e'_1 e_2}^n \mid e'_1 \in (update\ l\ e_1\ xs) \cap \mathbb{E} \} \cup (xs \cap \{err, div\})$

$update(\ell' \triangleleft l) \bigwedge_{e_1 e_2}^n xs = \{ \bigwedge_{e_1 e'_2}^n \mid e'_2 \in (update\ l\ e_2\ xs) \cap \mathbb{E} \} \cup (xs \cap \{err, div\})$

Figure 4.8: Helper functions

expressions where each element is obtained by replacing the sub-expression of e at the location l with an element of xs , with appropriate handling of errors and divergence.

Figure 4.9 shows the essential notations for defining the weakest precondition calculus. Since we will again have fixed-point operators in the weakest precondition calculus, we need to ensure that least fixed points exist, by operating in a domain which is again a cpo, and show that our wp function is monotone with respect to that domain. The ordering of our domain \mathfrak{D}_L is a point-wise lifted set ordering, of which the bottom element is the empty set.

Similar to the semantic environment introduced for the denotational semantics in figure 4.4, the logic environment contains mappings of (fixed point) variables to an element of our logic domain (which is a function). Since we mutually define weakest must succeed preconditions and weakest may error preconditions, a fixed-point variable can map to two different functions. We use the tags \cdot (must succeed) and \uparrow (may error) to distinguish these two different mappings.

With these notations and helper partial functions, we provide the location-based weakest precondition calculus. For presentation purposes, we simplify our definitions by only considering the cases where location l actually exists in the expression. In our Isabelle/HOL formalisation, we make this explicit in the definition of wp and wp^\uparrow .

$$\begin{array}{ll}
\text{Position } i := & \ell \mid \ell^* \qquad \text{Location}(\mathbb{L}) \quad l := \quad \epsilon \mid l \triangleright i \mid i \triangleleft l \\
\text{Variable}(\mathbb{V}) \quad X \ Y \ Z \dots & \qquad \text{Tag}(\mathbb{T}) \quad t := \quad \cdot \mid \uparrow \\
\\
\text{Logic Domain} \quad \mathfrak{D}_L = & \mathbb{L} \rightarrow \mathcal{P}(\mathbb{E}) \rightarrow \mathcal{P}(\mathbb{E}) \\
\text{Logic Environment}(\Gamma_L) \quad \zeta : & (\mathbb{V} \times \mathbb{T}) \rightarrow \mathfrak{D}_L
\end{array}$$

$$\begin{array}{ll}
wp_{\zeta:\Gamma_L \Vdash s:\mathbb{S}@l:\mathbb{L}}(P : \mathcal{P}(\mathbb{E})) : (R_w : \mathcal{P}(\mathbb{E})) & \text{(Weakest must succeed precondition)} \\
wp_{\zeta:\Gamma_L \Vdash s:\mathbb{S}@l:\mathbb{L}}^\uparrow(P : \mathcal{P}(\mathbb{E})) : (R_w : \mathcal{P}(\mathbb{E})) & \text{(Weakest may error precondition)}
\end{array}$$

Figure 4.9: Basic notations

Figure 4.10 shows the weakest preconditions for basic strategies: SKIP, ABORT and *atomic*. Trivially, the weakest must succeed precondition and weakest may error precondition for SKIP are the same, i.e., the given postcondition P , since the execution of SKIP never results in error or divergence, nor changes the input expression. As for ABORT, since it will always result in an error no matter what input expression is given, its weakest must succeed precondition is the empty set and its weakest may error precondition is the set of all expressions. The weakest preconditions of atomic strategies are defined using their denotational semantics (cf. figure 4.4): the weakest must succeed precondition is the set of input expressions, for each expression of which applying the atomic strategy to its sub-expression at the given location l should result in a (singleton) set of expressions which is a subset of the given postcondition P . The weakest may error postcondition is defined in a similar manner, the

$$\begin{array}{ll}
wp_{\zeta \Vdash \text{SKIP}@l}(P) = P & wp_{\zeta \Vdash \text{ABORT}@l}(P) = \emptyset \\
wp_{\zeta \Vdash \text{SKIP}@l}^\uparrow(P) = P & wp_{\zeta \Vdash \text{ABORT}@l}^\uparrow(P) = \mathbb{E} \\
\\
wp_{\zeta \Vdash \text{atomic}@l}(P) = \{e \mid (\llbracket \text{atomic} \rrbracket \emptyset(\mathfrak{h}_l e)) \sqsupseteq_l e \subseteq P\} \\
wp_{\zeta \Vdash \text{atomic}@l}^\uparrow(P) = \{e \mid (\llbracket \text{atomic} \rrbracket \emptyset(\mathfrak{h}_l e)) \sqsupseteq_l e \subseteq P \cup \{\text{err}\}\}
\end{array}$$

Figure 4.10: Location-based weakest preconditions for basic strategies

$$wp_{\zeta \Vdash s; t @ l}(P) = wp_{\zeta \Vdash s @ l}(wp_{\zeta \Vdash t @ l}(P)) \quad wp_{\zeta \Vdash s; t @ l}^\uparrow(P) = wp_{\zeta \Vdash s @ l}^\uparrow(wp_{\zeta \Vdash t @ l}^\uparrow(P))$$

(Sequential composition)

$$\begin{aligned} wp_{\zeta \Vdash s <+ t @ l}(P) &= wp_{\zeta \Vdash s @ l}(P) \cup (wp_{\zeta \Vdash s @ l}^\uparrow(P) \cap wp_{\zeta \Vdash t @ l}(P)) \\ wp_{\zeta \Vdash s <+ t @ l}^\uparrow(P) &= wp_{\zeta \Vdash s @ l}(P) \cup (wp_{\zeta \Vdash s @ l}^\uparrow(P) \cap wp_{\zeta \Vdash t @ l}^\uparrow(P)) \end{aligned}$$

(Left choice)

$$\begin{aligned} wp_{\zeta \Vdash s <+> t @ l}(P) &= (wp_{\zeta \Vdash t @ l}^\uparrow(P) \cap wp_{\zeta \Vdash s @ l}(P)) \cup (wp_{\zeta \Vdash s @ l}^\uparrow(P) \cap wp_{\zeta \Vdash t @ l}(P)) \\ wp_{\zeta \Vdash s <+> t @ l}^\uparrow(P) &= wp_{\zeta \Vdash s @ l}^\uparrow(P) \cap wp_{\zeta \Vdash t @ l}^\uparrow(P) \end{aligned}$$

(Nondeterministic choice)

Figure 4.11: Location-based weakest preconditions for combinators

only difference is that the resulting set of expressions should be a subset of $P \cup \{err\}$. It does not matter what semantic environment is given here when we invoke the semantics, so we just use the environment which maps all variables to $\{div\}$, denoted by \emptyset . Remember that the operators \mathfrak{h} and \boxRightarrow are *lookup* and *update*.

Figure 4.11 shows the weakest preconditions for combinators: sequential composition, left choice and nondeterministic choice. Intuitively, the weakest must succeed precondition of the sequential composition $s ; t$ is simply to sequentially compose the weakest must succeed preconditions of s and t where the post condition of s is the weakest must succeed precondition of t . The same approach is taken for defining the weakest may error precondition. The weakest must succeed precondition of the left choice $s <+ t$ is the union of the set of expressions that can be successfully rewritten by the strategy s and the set of expressions for which applying s may result in error but that can be successfully rewritten by the strategy t . Its weakest may error condition additionally includes the set of expressions for which applying the strategy t may result in error. The definitions of the weakest preconditions of the nondeterministic choice $s <+> t$ capture the angelic nondeterminism for *err* and demonic nondeterminism for *div*. Its weakest must succeed precondition is the set of expressions to which applying neither the strategy s nor t will diverge and which can be success-

$$\begin{aligned}
wp_{\zeta \Vdash one(s) @ l}^{\uparrow}(P) &= (wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P) \cap wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P)) \cup (wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P) \cap wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)) \\
wp_{\zeta \Vdash one(s) @ l}^{\uparrow}(P) &= \{e \mid (\mathfrak{h}_l e) = Leaf\} \cup (wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P) \cap wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P))
\end{aligned}$$

(One)

$$\begin{aligned}
wp_{\zeta \Vdash some(s) @ l}^{\uparrow}(P) &= wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P)) \cup wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)) \\
&\cup (wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P) \cap wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P))) \\
&\cup (wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P) \cap wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)))
\end{aligned}$$

$$\begin{aligned}
wp_{\zeta \Vdash some(s) @ l}^{\uparrow}(P) &= \{e \mid (\mathfrak{h}_l e) = Leaf\} \\
&\cup wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P)) \cap wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)) \\
&\cap (wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P) \cup wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P))) \\
&\cap (wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P) \cup wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)))
\end{aligned}$$

(Some)

$$\begin{aligned}
wp_{\zeta \Vdash all(s) @ l}^{\uparrow}(P) &= (P \cap \{e \mid (\mathfrak{h}_l e) = Leaf\}) \\
&\cup wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P)) \cup wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)) \\
wp_{\zeta \Vdash all(s) @ l}^{\uparrow}(P) &= (P \cap \{e \mid (\mathfrak{h}_l e) = Leaf\}) \\
&\cup (wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(P)) \cap wp_{\zeta \Vdash s @ l \triangleright r}^{\uparrow}(wp_{\zeta \Vdash s @ l \triangleright \ell}^{\uparrow}(P)))
\end{aligned}$$

(All)

Figure 4.12: Location-based weakest preconditions for traversals

fully rewritten by at least one of s and t . The weakest may error precondition relaxes this last requirement by including the set of expressions to which applying both s and t may result in an error.

Location is very important for defining the weakest preconditions of traversals. Demonstrated in figure 4.12, the approach of defining the weakest preconditions for $one(s)$ is again similar to nondeterministic choice, as $one(s)$ nondeterministically chooses one of the left or right child of the current expression to apply the strategy s to. Its weakest must succeed precondition is a set of expressions that are not leaf nodes. For each of them, applying s to either its left child or right child should not diverge, and at least one of its children must be successfully rewritten by s . The

weakest may error precondition of $one(s)$ includes all expressions that are leaf nodes as well as expressions whose both children to which applying s may result in error. The weakest must succeed precondition of $some(s)$ is a set of expressions that are not leaf nodes. For each of them, if the given strategy s can be applied to *both* of its children successfully, the result of applying s to both of them regardless of the ordering of the application should satisfy the postcondition P . In addition, applying s to one of its children may result in an error, but not for both of its children. Again, expressions with children to which applying s diverges are excluded from the weakest must succeed precondition. Similar to $one(s)$, the weakest may error preconditions includes all leaf expressions and expressions whose both children to which applying s may result in error. Since $all(s)$ requires the strategy s to be applied to either a leaf expression or both children of an expression which is not a leaf, intuitively, its weakest must succeed precondition is a set of leaf expressions, or expressions of which both children can be successfully rewritten by the strategy s regardless of the order of the application of s . Its weakest may error precondition again includes all leaf expressions and expressions with children to which applying s may result in an error.

Lastly, we introduce the weakest preconditions for the fixed-point operator, shown in figure 4.13, which are defined using simultaneous induction. Δ contains a pair of simultaneously defined least fixed points \mathcal{X} and \mathcal{Y} which are used to define the weakest must succeed precondition and weakest may fail precondition respectively. In these fixed-point equations, we extend the logic environment ζ with mappings from the fixed-point variable with tags (X, \cdot) and (X, \uparrow) to the least fixed points \mathcal{X} and \mathcal{Y} respectively.

The weakest must succeed precondition a (fixed point) variable X is calculated by applying the function obtained by looking up (X, \cdot) in the logic environment ζ to the location l and postcondition P . For the weakest may fail precondition, the function applied to l and P is obtained by looking up X with the may fail tag \uparrow from ζ .

4.4.3 The Soundness of the Weakest Precondition Calculus w.r.t. the Formal Semantics

Since our weakest precondition calculus is designed to reason about the execution of strategies, it is essential to prove it is *sound* with respect to the formal semantics introduced in section 4.3. Specifically, we define the soundness of the weakest must succeed precondition as theorem 4.4.1, and the soundness of the weakest may error

$$\begin{aligned}
wp_{\zeta \Vdash X @ l}(P) &= \zeta(X, \cdot) l P \quad (\text{where } \zeta(X, \cdot) \text{ def.}) \\
wp_{\zeta \Vdash X @ l}^\uparrow(P) &= \zeta(X, \uparrow) l P \quad (\text{where } \zeta(X, \uparrow) \text{ def.}) \\
&\quad (\text{Fixed-point variable}) \\
wp_{\zeta \Vdash \mu X.s @ l}(P) &= [\text{LFP } \mathcal{X} : \Delta] l P \quad wp_{\zeta \Vdash \mu X.s @ l}^\uparrow(P) = [\text{LFP } \mathcal{Y} : \Delta] l P \\
\text{Where : } \Delta &= \begin{cases} \mathcal{X} l P &= wp_{\zeta[(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{Y}]} \Vdash s @ l(P) \\ \mathcal{Y} l P &= wp_{\zeta[(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{Y}]} \Vdash s @ l(P) \end{cases} \\
&\quad (\text{Fixed-point operator})
\end{aligned}$$

Figure 4.13: Location-based weakest preconditions for fixed-point operators

precondition as theorem 4.4.2. Both of these theorems have the same assumption to relate the logic and semantic environments ζ and ξ . This assumption states that given any variable X , location l and postcondition P , executing the function obtained by looking up X in the logic environment ζ — with the must succeed tag or the may error tag correspondingly — gives the set of expressions, at the location l of each of which executing the semantics of the variable ($\xi(X)$) results in a subset of the postcondition P or $P \cup \{err\}$ respectively. From this assumption, theorem 4.4.1 concludes that the weakest must succeed precondition $wp_{\zeta \Vdash s @ l}(P)$ should equal to the set of expressions on which executing the semantics of s gives a subset of P . Similarly, theorem 4.4.2 says that under the same assumptions, the weakest may error precondition $wp_{\zeta \Vdash s @ l}^\uparrow(P)$ should equal to the set of expressions on which executing the semantics of s gives a subset of $P \cup \{err\}$.

Theorem 4.4.1 (Soundness theorem for Weakest Must Succeed Precondition).

$$\begin{aligned}
&\forall X l P. \zeta(X, \cdot) l P = \{e \mid \xi(X)(\mathfrak{h}_l e) \sqsupseteq_l e \subseteq P\} \\
&\wedge \zeta(X, \uparrow) l P = \{e \mid \xi(X)(\mathfrak{h}_l e) \sqsupseteq_l e \subseteq P \cup \{err\}\} \\
\hline
&wp_{\zeta \Vdash s @ l}(P) = \{e \mid (\llbracket s \rrbracket \xi(\mathfrak{h}_l e)) \sqsupseteq_l e \subseteq P\}
\end{aligned}$$

Theorem 4.4.2 (Soundness theorem for Weakest May Error Precondition).

$$\begin{aligned}
&\forall X l P. \zeta(X, \cdot) l P = \{e \mid \xi(X)(\mathfrak{h}_l e) \sqsupseteq_l e \subseteq P\} \\
&\wedge \zeta(X, \uparrow) l P = \{e \mid \xi(X)(\mathfrak{h}_l e) \sqsupseteq_l e \subseteq P \cup \{err\}\} \\
\hline
&wp_{\zeta \Vdash s @ l}^\uparrow(P) = \{e \mid (\llbracket s \rrbracket \xi(\mathfrak{h}_l e)) \sqsupseteq_l e \subseteq P \cup \{err\}\}
\end{aligned}$$

We prove these two theorems simultaneously, by induction on the strategy s . For the fixed-point operator cases, we make use of Scott induction. The proof is mechanised in Isabelle/HOL.

4.5 Reasoning About Strategies with Weakest Precondition Calculus

As discussed in section 4.2, there are some strategies that can never be executed successfully, such as strategies that always diverge like $repeat(SKIP)$ and strategies that are not well composed like $mult_{com} ; add_{com}$. We call such strategies *bad* strategies. Formally, we define *good* and *bad* strategies in terms of our weakest precondition calculus as definition 4.5.1 and definition 4.5.2, where the formal definition of bad strategies is the negation of good strategies.

Definition 4.5.1 (Good strategies). *A strategy s is good iff for a given postcondition P :*

$$wp_{\zeta \Vdash s @ l}(P) \neq \emptyset$$

Definition 4.5.2 (Bad strategies). *A strategy s is bad iff for a given postcondition P :*

$$wp_{\zeta \Vdash s @ l}(P) = \emptyset$$

For strategies that can terminate and are well composed, they may not be able to successfully rewrite any input expression into an expression satisfying our desired postcondition. For instance, even though the atomic strategy add_{com} is a good strategy, applying it to $3 * 4$ would result in an error. Also, as illustrated in section 4.2, when encoding a normalisation strategy for rewriting an input lambda expression into its $\beta\eta$ -normal form, such strategy can diverge on some input expressions (e.g., the expression Ω given below). If it does terminate on an input expression, it ought to rewrite all reducible sub-expressions of such input expression. We formally define the successful executions and unsuccessful executions of good strategies as definition 4.5.3 and definition 4.5.4.

Definition 4.5.3 (Successful execution). *An execution of a good strategy s , on an input expression e is successful iff for a given postcondition P :*

$$e \in wp_{\zeta \Vdash s @ l}(P) \quad (\text{where: } wp_{\zeta \Vdash s @ l}(P) \neq \emptyset)$$

Definition 4.5.4 (Unsuccessful execution). *An execution of a good strategy s on an input expression e is unsuccessful iff for a given postcondition P :*

$$e \notin wp_{\zeta \Vdash s @ l}(P) \quad (\text{where: } wp_{\zeta \Vdash s @ l}(P) \neq \emptyset)$$

Next, we demonstrate how to use the location-based weakest precondition calculus to reason about the execution of strategies. All examples we discuss are mechanised in Isabelle/HOL.

4.5.1 Reasoning About Termination

Strategies can diverge. Recall from section 4.2 that $repeat(s)$ is defined as $\mu X. try(s; X)$ where $try(s)$ is defined as $s <+> SKIP$. We can derive the weakest precondition formula of $repeat(s)$ by the weakest precondition formulae of $SKIP$, left choice, sequential composition and the fixed-point operator:

$$wp_{\zeta \Vdash repeat(s) @ l}(P) = wp_{\zeta \Vdash repeat(s) @ l}^{\uparrow}(P) = [LFP \mathcal{X} : \Delta] \mid P$$

where Δ is the fixed-point equation

$$\mathcal{X} \mid P = wp_{\zeta [(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{X}]} \Vdash s @ l(\mathcal{X} \mid P) \cup (P \cap wp_{\zeta [(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{X}]} \Vdash s @ l}^{\uparrow}(\mathcal{X} \mid P))$$

Although the execution of $repeat(s)$ would never result in an error since its weakest may error precondition formula is identical to its weakest must succeed precondition, it may diverge.

A simple example of a diverging strategy we have introduced is the strategy $repeat(SKIP)$. It is straightforward to conclude that it is a bad strategy using the weakest precondition calculus. With the weakest must succeed precondition formulae of $repeat(s)$ and $SKIP$, we calculate that for the set of all expressions as the post condition, the weakest must succeed precondition of $repeat(SKIP)$ is an empty set:

$$wp_{\zeta \Vdash repeat(SKIP) @ \epsilon}(\mathbb{E}) = \emptyset$$

Intuitively, such a result indicates that there is no expression that can be successfully rewritten by the strategy $repeat(SKIP)$. According to the definition 4.5.2, we can conclude that the diverging strategy $repeat(SKIP)$ is bad strategy.

Since we apply demonic nondeterminism on divergence as discussed in section 4.4, the strategy $SKIP <+> repeat(SKIP)$ always diverges. To show that it is a bad strategy, we can again calculate its weakest must succeed precondition with the set of all

expressions as the postcondition:

$$\mathcal{WP}_{\zeta \Vdash \text{SKIP} <+ \text{repeat}(\text{SKIP}) @ \epsilon}(\mathbb{E}) = \emptyset$$

Again, we obtain an empty set as its weakest must succeed precondition, indicating that such a strategy can never be successfully executed on any input expression.

Strategies that can terminate are potentially good strategies. For instance, the strategy $\text{SKIP} <+ \text{repeat}(\text{SKIP})$ always terminates. To conclude it being a good strategy, we calculate its weakest must succeed precondition:

$$\mathcal{WP}_{\zeta \Vdash \text{SKIP} <+ \text{repeat}(\text{SKIP}) @ \epsilon}(\mathbb{E}) = \mathbb{E}$$

Intuitively, because left choice prioritises the strategy on the left hand side of the combinator over the strategy on the right hand side, SKIP is always preferred over $\text{repeat}(\text{SKIP})$ here. Therefore, $\text{SKIP} <+ \text{repeat}(\text{SKIP})$ always terminates and produces expressions. According to the definition 4.5.1, we conclude that the terminating strategy $\text{SKIP} <+ \text{repeat}(\text{SKIP})$ is a good strategy.

4.5.2 Reasoning About Well Composed Strategies

Strategies that terminate may still not be good strategies, since they can be not well composed and always result in error. An example of a not well composed strategy that we have introduced in section 4.2 is $\text{mult}_{\text{com}} ; \text{add}_{\text{com}}$. According to the weakest precondition formulae for atomic strategies and the sequential composition presented in figure 4.10 and figure 4.11, we calculate its weakest must succeed precondition for the set of all expressions as the postcondition:

$$\mathcal{WP}_{\zeta \Vdash \text{mult}_{\text{com}} ; \text{add}_{\text{com}} @ \epsilon}(\mathbb{E}) = \emptyset$$

Since its weakest must succeed precondition is an empty set, with definition 4.5.2, we can conclude that the strategy $\text{mult}_{\text{com}} ; \text{add}_{\text{com}}$ is a bad strategy.

Well composed terminating strategies are good strategies. For example, given an atomic strategy add_{id} defined as:

$$\text{add}_{\text{id}} : 0 + a \rightsquigarrow a$$

The strategy $\text{add}_{\text{com}} ; \text{add}_{\text{id}}$ is a well composed strategy. In practice, it can successfully rewrite an expression $3 + 0$ into the expression 3. We are able to conclude that the

$$\begin{array}{l} \text{Lambda Expression} \quad e := \quad Id \iota \mid \overset{Abs}{\bullet \over e} \mid \overset{App}{e \over e} \\ \\ \text{Index} \quad \iota \in \mathbb{N} \end{array}$$

Figure 4.14: The syntax of the lambda calculus

strategy $add_{com} ; add_{id}$ is a good strategy again by checking its weakest must succeed precondition for the set of all expressions as the postcondition:

$$wp_{\zeta \Vdash add_{com} ; add_{id} @ \epsilon}(\mathbb{E}) = \{e \mid e = a + 0\}$$

Since the calculated weakest must succeed precondition is not an empty set, according to the definition 4.5.1, the strategy $add_{com} ; add_{id}$ is a good strategy.

4.5.3 Reasoning About Beta-Eta Normalisation

In section 4.2, we have defined the normalise strategy by composing the strategy $repeat(s)$ and the top-down traversal $topDown(s)$ as $normalise(s) = repeat(topDown(s))$, which keeps applying a given strategy s to every possible sub-expression of an expression until s is no longer applicable.

One example usage of the normalisation strategy we demonstrated is to reduce an expression in λ -calculus into the $\beta\eta$ -normal form. Given the β -reduction and η -reduction as two atomic strategies $beta$ and eta , we can express the strategy for calculating the $\beta\eta$ -normal form as:

$$BENF = normalise(beta <+ eta)$$

Furthermore, we define a predicate to assert that an expression is in $\beta\eta$ -normal form, simply by stating that the $beta$ and eta atomic strategies must not be defined for any location in the expression:

$$isBENF \ e \Leftrightarrow \forall l. \ beta(\hbar_l \ e) \mathbf{undef} \wedge \ eta(\hbar_l \ e) \mathbf{undef} \quad (\text{where: } \hbar_l \ e \text{ is defined})$$

It is well known that not every λ -expression has such a normal form. With our location-based weakest precondition calculus, we are able to reason about whether an expression can be normalised by the strategy $BENF$ into a $\beta\eta$ -normal form.

Firstly, in figure 4.14, we provide an encoding of the lambda calculus with de Bruijn indices using the expression tree structure we introduced, which takes the

form of either a *Leaf* or a node \widehat{e}^n_e . Specifically, we encode an *Id* expression (a de Bruijn index) as a *Leaf* and both an abstraction and an application as nodes. Then we encode beta reduction and eta reduction as two atomic strategies:

$$\begin{array}{ccc} \text{beta:} & \begin{array}{c} \text{App} \\ \swarrow \quad \searrow \\ \text{Abs} \quad e \\ \swarrow \quad \searrow \\ \bullet \quad f \end{array} & \rightsquigarrow f[e/0] \end{array} \quad \begin{array}{ccc} \text{eta:} & \begin{array}{c} \text{Abs} \\ \swarrow \quad \searrow \\ \bullet \quad \text{App} \\ \quad \swarrow \quad \searrow \\ \quad f \quad \text{Id } 0 \end{array} & \rightsquigarrow f \downarrow_0 \end{array}$$

where $f[e/0]$ is the de Bruijn substitution of the index 0 with the expression e in f and $f \downarrow_0$ is the de Bruijn down shifting eliminating the index 0 in f .

Next we introduce the weakest precondition formula for the strategy *normalise*(s), which is calculated using the weakest precondition formulae of *repeat*(s) (introduced in section 4.5.1) and *topDown*(s). Recall that in section 4.2 the strategy *topDown*(s) is defined using the left choice combinator, the traversal *one*(s) as well as the fixed-point operator:

$$\text{topDown}(s) = \mu X. (s <+ \text{one}(X))$$

We can derive its weakest must succeed precondition and weakest may error precondition formulae:

$$\text{wp}_{\zeta \Vdash \text{topDown}(s) @ l}(P) = [\text{LFP } \mathcal{X} : \Delta] \text{ } l P \quad \text{wp}_{\zeta \Vdash \text{topDown}(s) @ l}^\uparrow(P) = [\text{LFP } \mathcal{Y} : \Delta] \text{ } l P$$

Where:

$$\Delta = \begin{cases} \mathcal{X} \text{ } l P &= \text{wp}_{\zeta \Vdash [(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{Y}]} \Vdash s @ l(P) \cup (\text{wp}_{\zeta \Vdash [(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{Y}]} \Vdash s @ l}^\uparrow(P) \\ &\cap ((\mathcal{Y} (l \triangleright \ell) P \cap \mathcal{X} (l \triangleright \nu) P) \cup (\mathcal{Y} (l \triangleright \nu) P \cap \mathcal{X} (l \triangleright \ell) P)) \\ \mathcal{Y} \text{ } l P &= \text{wp}_{\zeta \Vdash [(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{Y}]} \Vdash s @ l(P) \cup (\text{wp}_{\zeta \Vdash [(X, \cdot) \mapsto \mathcal{X}, (X, \uparrow) \mapsto \mathcal{Y}]} \Vdash s @ l}^\uparrow(P) \\ &\cap (\mathcal{Y} (l \triangleright \ell) P \cap \mathcal{Y} (l \triangleright \nu) P) \end{cases}$$

With the weakest precondition formulae for *topDown*(s) defined, we can subsequently provide the weakest precondition formula for the strategy *normalise*(s). Note that its weakest must succeed precondition and weakest may error precondition share the same formula, just like *repeat*(s):

$$\text{wp}_{\zeta \Vdash \text{normalise}(s) @ l}(P) = \text{wp}_{\zeta \Vdash \text{normalise}(s) @ l}^\uparrow(P) = [\text{LFP } \mathcal{X}_r : \Delta_r] \text{ } l P$$

Where:

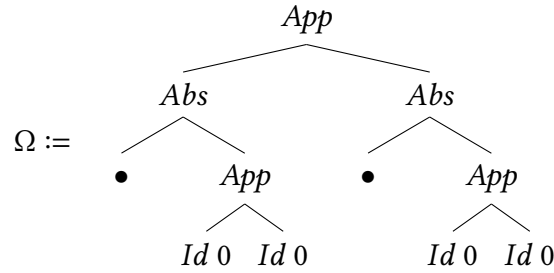
$$\Delta_r = \mathcal{X}_r \text{ } l P = [\text{LFP } \mathcal{X}_t : \Delta_t] \text{ } l P \cup (([\text{LFP } \mathcal{Y}_t : \Delta_t] \text{ } l P) \cap P)$$

$$\Delta_t = \begin{cases} \mathcal{X}_t l P &= wp_{\zeta}[(X, \cdot) \mapsto \mathcal{X}_r, (X, \uparrow) \mapsto \mathcal{X}_r, (Y, \cdot) \mapsto \mathcal{X}_t, (Y, \uparrow) \mapsto \mathcal{Y}_t] \Vdash_{s@l} (\mathcal{X}_r l P) \\ &\cup (wp_{\zeta}^{\uparrow}[(X, \cdot) \mapsto \mathcal{X}_r, (X, \uparrow) \mapsto \mathcal{X}_r, (Y, \cdot) \mapsto \mathcal{X}_t, (Y, \uparrow) \mapsto \mathcal{Y}_t] \Vdash_{s@l} (\mathcal{X}_r l P) \\ &\cap ((\mathcal{Y}_t(l \triangleright \ell) P \cap \mathcal{X}_t(l \triangleright \ell) P) \cup (\mathcal{Y}_t(l \triangleright \ell) P \cap \mathcal{X}_t(l \triangleright \ell) P))) \\ \mathcal{Y}_t l P &= wp_{\zeta}[(X, \cdot) \mapsto \mathcal{X}_r, (X, \uparrow) \mapsto \mathcal{X}_r, (Y, \cdot) \mapsto \mathcal{X}_t, (Y, \uparrow) \mapsto \mathcal{Y}_t] \Vdash_{s@l} (\mathcal{X}_r l P) \\ &\cup (wp_{\zeta}^{\uparrow}[(X, \cdot) \mapsto \mathcal{X}_r, (X, \uparrow) \mapsto \mathcal{X}_r, (Y, \cdot) \mapsto \mathcal{X}_t, (Y, \uparrow) \mapsto \mathcal{Y}_t] \Vdash_{s@l} (\mathcal{X}_r l P) \\ &\cap (\mathcal{Y}_t(l \triangleright \ell) P \cap \mathcal{Y}_t(l \triangleright \ell) P)) \end{cases}$$

With the weakest precondition formula for *normalise(s)*, we can first conclude that the strategy *BENF* for calculating the $\beta\eta$ -normal form for expressions is a good strategy by showing:

$$wp_{\zeta} \Vdash_{BENF@l} (\mathbb{E}) \neq \emptyset$$

Although the strategy *BENF* is good, some expressions are not able to be rewritten by it to a $\beta\eta$ -normal form. For instance, the expression Ω is defined as:



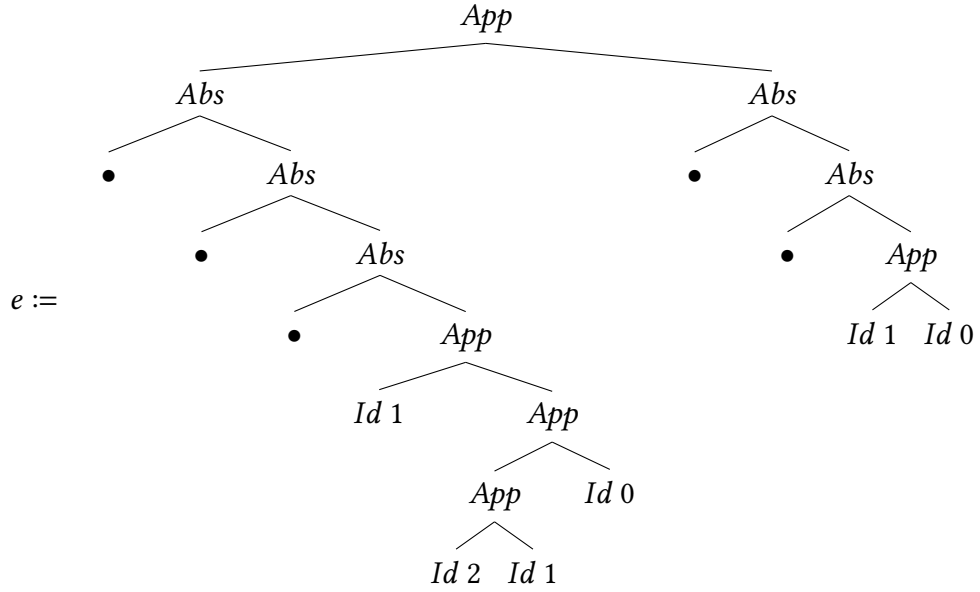
Applying the strategy *BENF* to the expression Ω will diverge, namely, the execution of the strategy *BENF* on Ω is unsuccessful. We draw this conclusion by showing that Ω is not an expression in the weakest must succeed precondition of *BENF* no matter what the postcondition is:

$$\Omega \notin wp_{\zeta} \Vdash_{BENF@e} (\mathbb{E})$$

We prove this proposition straightforwardly using Scott induction.

Beside identifying expressions that fail to be normalised into a $\beta\eta$ -normal form using *BENF*, we are also interested in examining whether a complex expression is indeed rewritten into a $\beta\eta$ -normal form after applying the strategy *BENF*. For instance,

given an expression e defined as:



we show that applying the strategy $BENF$ to the expression e does rewrite it to a $\beta\eta$ -normal form by showing the proposition below holds:

$$e \in wp_{\zeta \Vdash BENF @ \epsilon}(\{e \mid isBENF\ e\})$$

The proof of this proposition is also straightforward, merely requiring the repeated unfolding of fixed-point operators. On the basis of this result, we can conclude that the strategy $BENF$ performs the rewrite on the input expression e as we expected, namely, rewriting e into its $\beta\eta$ -normal form.

4.5.4 Discussion

As this section demonstrates, our formal calculus provides precise description of strategies, independent of their length and complexity. It also provides a good characterisation of desired properties to be satisfied after the execution of a strategy, as well as of expressions that can be successfully rewritten. Additionally, our calculus is capable of performing non-trivial reasoning about rewrite strategies. Specifically, the reasoning about beta-eta normalisation already features strategy combinators, traversals and recursion: the fundamental ingredients of strategic rewriting. As our framework is fully mechanised in Isabelle/HOL, reasoning can be performed directly in and facilitated by the proof assistant. Therefore, it is conceivable — still with a significant effort — to use our framework for reasoning about complex applications,

including Elevate (Hagedorn et al., 2020) compiler optimisations. A significant initial hurdle is to encode the language that is rewritten (e.g. the lambda calculus in section 4.5.3) as well as application-specific rewrites in Isabelle/HOL, before we can start reasoning about the behaviour of more complex rewrite strategies. With our formal calculus and its Isabelle/HOL implementation it would be possible to build up a library of standard languages and rewrites, to facilitate reasoning about increasingly complex practical applications.

4.6 Related Work

Strategic Rewriting and Traversals Term rewriting systems (Dershowitz, 1985) are a powerful and versatile method to express syntactic transformations. Strategic rewriting languages, which give programmers control over the rewriting process, have seen applications in many areas. Initial efforts, such as the language ELAN (Borovanský et al., 1996), focused on using rewriting as a way to model deduction and computation. The previously mentioned Stratego (Visser et al., 1998; Visser, 2001; Bravenboer et al., 2008), which uses System S as its core language, is designed for developing language interpreters in the Spoofox Language Workbench (Wachsmuth et al., 2014). Elevate (Hagedorn et al., 2023, 2020) is very much in the style of Stratego, but is instead targeted towards guiding optimisations in a compiler for high performance computing. The language TL (Winter and Subramaniam, 2004) applies strategic rewriting to data processing tasks, and Strafunski (Lämmel and Visser, 2002), which is again a Stratego-like language, uses strategies for datatype-generic programming. Traversals are an essential feature of System S that also appear in other program transformation designs, such as the ‘Scrap your boilerplate’ (SYB) style traversals (e.g. `everywhere`, `everything`, `anyDescendant`, `anyAncestor` etc.) for XML programming (Lämmel, 2007). Reachability constraints are added to types of these traversals for detecting queries that result in an empty set and transformations that always fail or do not change anything. To analyse strategic programs some algebraic laws are discussed by Cunha and Visser (2007) for equational reasoning and by Lämmel et al. (2013) as hints of potential dead code. One could potentially make use of our weakest precondition calculus to prove and generalise these laws.

Weakest Preconditions Weakest preconditions were introduced by Dijkstra (Dijkstra, 1975). Bonsangue and Kok (1992) extend Dijkstra’s calculus to include recur-

sion in the same way that we do. Weakest preconditions are key to Cook’s proof (Cook, 1978) of the relative completeness of Floyd-Hoare Logic (Floyd, 1967; Hoare, 1969b), and are similarly used by Goncharov and Schröder (2013) to show relative completeness of their Hoare Logic for programs with monadic effects. Morgan (1994) uses weakest preconditions as the semantic foundation for his refinement calculus, enabling stepwise derivation of programs from their specifications. In recent work, Aguirre et al. (2022) explore the categorical structure of compositional weakest preconditions, characterising them as those that are obtained from the Cartesian lifting of some monad. As a related application of weakest preconditions, Swierstra and Baaen (2019) provide a weakest prediction semantics for effectful programs, accounting for exceptions, state, non-determinism and general recursion. Their work could possibly be an alternative approach to achieve some of the goals of our work, although the application of such a formalism to the form of rewriting in formalisms like system S is not immediate. For example, it is unclear whether System S with its handling of errors and non-termination would actually form a monad. Errors alone can, of course, be handled by the Error monad; the interaction with divergence and errors is more sophisticated. As a consequence, this may give rise to complications of a similar order of magnitude as the ones addressed in this chapter.

Existing Formalisation and Verification We are not the first to examine strategic rewriting languages formally. Both the initial paper on Stratego (Visser et al., 1998) and the paper on System S (Visser and Benaissa, 1998) present big-step operational semantics. However these semantics do not model divergence, and are not the basis for any formal claims. In this work, by contrast, we model all possible outcomes including divergence denotationally, and we show the denotational model equivalent to an extended big-step operational semantics of System S that includes divergence, by establishing the computational soundness and adequacy with respect to the extended big-step operational semantics. Kaiser and Lämmel (2009) formalise a subset of System S without divergence in Isabelle/HOL by shallow embedding, but this formalisation does not include the general fixed-point operator of System S, and the choice to use shallow embedding, while convenient for some tasks, precludes the formalisation of general, meta-theoretic properties about all strategies. In our formalisation, we opt for a deep embedding, enabling us to mechanise all of the definitions and proofs in this chapter. Focusing on traversals in strategic languages, Lämmel et al. (2013) characterise a list of strategic programming errors and discuss ways to

avoid these errors with static typing and static analysis. With a different approach, we provide a general and formal characterisation of “good” and “bad” strategies as well as successful and unsuccessful executions of strategies, using our location-based weakest precondition calculus.

Kieburtz (2001), an inspiration for this work, informally sketches some weakest precondition rules for Stratego. Rather than a location-based weakest precondition calculus such as ours, Kieburtz (2001) includes assertions in modal logic (specifically a combination of CTL and the modal μ -calculus), where the various tree modalities allow movement to different sub-expressions. However, this modal logic variant does not have the expressive power of our framework because of our choice of location language. For instance, CTL is not expressive enough to reason about the one operator. When it comes to traversals, Kieburtz (2001) does not define general predicate transformers for modal assertions, and thus Kieburtz’s (2001) rules do not form a complete calculus. It is not clear how Kieburtz’s (2001) approach could be extended to handle traversals in their full generality. In our work, our assertions are just sets of expressions, and we move around an expression by associating locations to our weakest preconditions. This enables us to define general rules for traversals, allowing a compositional and complete calculus for all strategies and all postconditions. In addition, the fixed-point operator is not well constructed in Kieburtz’s (2001) work and it is not proven to be monotone, whereas we have a correct treatment of the fixed-point operator and have proven monotonicity of all our formulae. Also, in Kieburtz’s (2001) work, soundness is not proven, whereas we prove the soundness of our weakest precondition calculus w.r.t. the formal semantics. Lastly, we provide a careful treatment of divergence with mutually defined wp and wp^\uparrow , while such a feature is not reflected in Kieburtz’s (2001) work.

Type Systems for Strategic Rewriting Languages A related but parallel strand of work is in giving *types* to strategic rewriting languages. Smits and Visser (2020) add gradual typing to Stratego and use it to find bugs in their strategies for language interpreters. Koppel (2023) uses typed strategies to model multi-language program transformations, Lämmel (2003) adds types to strategies with applications to generic programming in typed languages and Fu et al. (2023) makes use of structural typing with traces for checking ill-composed strategies statically. These type systems emphasise lightweight static or the hybrid of dynamic and static checking to find bugs, whereas our focus is on a complete semantic accounting of rewriting strategies, and

the development of a weakest precondition calculus that can demonstrate the absence of bugs, not merely their presence.

Kleene Algebra Strategic rewriting languages resemble a Kleene Algebra (Kozen, 1991) extended with traversals and a biased choice operator. There have been many other extensions to Kleene Algebra, most notably Concurrent Kleene Algebra (Hoare et al., 2011), which adds parallel composition, and Kleene Algebra with Tests (Kozen, 1997), which adds Boolean guards to model the semantics of **while** programs. Kozen (1999) shows that reasoning by Kleene Algebra with Tests entirely subsumes Hoare Logic for **while** programs. A version of Kleene Algebra with Tests, NetKAT, has been used to reason about packet switching networks (Anderson et al., 2014). Recently, Concurrent Kleene Algebra and NetKAT have been combined for reasoning about concurrent network systems (Wagemaker et al., 2022).

Denotational semantics and adequacy The appeal of the Scott-Strachey approach to semantics (Stoy, 1985) is in its local and compositional reasoning, and over the last 50 years it has been used for many diverse programming languages. As far as programming language abstractions go, the strategic rewriting language we consider is mostly standard, and we were able to use the following relevant semantic tools with relatively minor modification. Plotkin pioneered the powerdomain construction (1976) and later characterised it as the free semilattice over a domain (Hennessy and Plotkin, 1979). Most denotational accounts include an adequacy proof, and it is possible to prove them wholesale for standard programming languages with a myriad of expressive features (Simpson, 2004; Plotkin and Power, 2001; Johann et al., 2010). We found the decomposition of computational adequacy into dual inductive and coinductive arguments interesting, and we hope it could inform other reflective accounts of adequacy (Devesas Campos and Levy, 2018).

4.7 Conclusion and Future Work

We have presented Shoggoth, a rigorous formal foundation for strategic rewriting languages, including a comprehensive semantic accounting of System S, and a weakest precondition calculus to facilitate formal reasoning about rewriting strategies. Our semantic treatment models all possible executions of strategies including divergences in both denotational and big-step operational models, and our proofs of sound-

ness and adequacy demonstrate the equivalence of these models. Our location-based weakest precondition calculus is the first formal axiomatic treatment of rewriting strategies, and enables reasoning about traversals by having the notion of location for indicating where in an expression a given strategy operates. Our soundness proof justifies our location-based weakest precondition calculus with respect to our semantic models, and we demonstrate practical application of this calculus by applying it to realistic examples. All of our work has been mechanised in over 5,000 lines of Isabelle/HOL proof script.

Weakest precondition calculi form the basis of *verification condition generators* (VCGs), which are a key component of many automatic and semi-automatic verification tools such as VCC (Cohen et al., 2009) and Dafny (Leino, 2010), as well as of static analysers such as the popular Extended Static Checking extension for Java (Flanagan et al., 2002; Leino, 2005). We envision that our weakest precondition calculus could similarly inform the design of a VCG for automatic verification or static checking of rewriting strategies. We intend to use Shoggoth as a foundation for the development of tools for verification and, potentially, *synthesis* of rewriting strategies.

Epilogue

In this study, we have analysed and designed a reasoning framework for syntactic transformations and their compositions via processing their formal semantics. In the end of this chapter, I would like to enclose this chapter again with some high-level observations and a reflection.

In a strategic rewriting system, the syntax for manipulating expressions and semantics of the evaluation of expressions are interdependent. Since the syntactic transformations of expressions encode the process of evaluating the semantics of these expressions and by composing these syntactic transformation steps, we have the syntax of a strategic rewriting language, providing a concise interface to not only compose but also control the application of these strategies. Again, by analysing the semantics of such syntax for composing strategies, we are able to understand and reason about the execution of these compositions of syntactic transformations. One may find such a process of detailed analysis of some concise language constructions for syntactic transformations at odds with the intuition of reasoning about programs — normally people utilise some notions like type systems and logic formulae which are more abstract than the encoding of these programs. However, I find it is hard to design some constructs which are even more abstract than the existing syntax of these composi-



tions for reasoning about their executions, rather, the behaviours hidden beneath the syntactic constructs of these compositions are surprisingly complicated and can only be properly assessed by modelling their executions.

In modelling the semantics of System S, there is again a trade-off between concise abstractions and precise expressiveness. To be able to see the composition of strategies clearly, we simplified our understanding of atomic strategies, abstracting their complex implementation details by modelling them as partial functions. With such an abstraction, some computation details of these atomic strategies are not considered like the side effects of the execution of an atomic strategy. However, by doing so we are able to concentrate on what possible results they can produce contributing to the analysis of the compositions rather than how they get evaluated to produce these results, allowing us to separate the process of semantic analysis of the composition of strategies from the detailed constructions of the expressions to be rewritten, thus making the semantics and reasoning framework we have built more general.

Chapter 5




The Thorn and The Bird: Still We Do It

Conclusion



The bird with the thorn in its breast, it follows an immutable law; it is driven by it knows not what to impale itself, and die singing. At the very instant the thorn enters there is no awareness in it of the dying to come; it simply sings and sings until there is not the life left to utter another note. But we, when we put the thorns in our breasts, we know. We understand. And still we do it. Still we do it.

— Colleen McCullough “The Thorn Birds”



THROUGHOUT this thesis, three conceptual questions have been addressed via three studies presented in chapter 2, chapter 3, and chapter 4, ultimately, they all relate to the fundamental motivation of my researches concerning the meaning of computer programs — to design better abstraction for modelling and understanding programs as well as better formal frameworks for reasoning about programs.

The study discussed in chapter 2 is conducted to address the first conceptual question: How to design a better abstraction mechanism that allows programmers to effectively express *what* they want a computer to do via some declarative yet accurate *specifications* instead of *how* a computer should accomplish a task via some concrete *implementations*? The study specifically focuses on designing property-based container types in programming languages. In particular, we investigate ways to declar-

actively specify the properties of container types using formal specifications instead of having these properties concretely implemented for container types, allowing concrete implementations to be inferred from the specifications. In this study, we utilise existing verification techniques including formal specifications, data refinement, and refinement types for the purpose of designing declarative yet accurate abstractions. We demonstrate that these specifications describing *what* properties, especially properties giving an account of functional requirements that a container type and its operations should satisfy, which are separated from concrete container implementations describing *how* properties are satisfied. In terms of addressing the conceptual question regarding to the understanding of the relationship between specifications and implementations, the design of property based container types in this study indicates that declarative specifications enable better automation and optimisation for application programmers when selecting desired container implementations in programs.

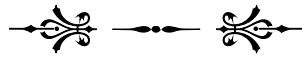
The study discussed in chapter 3 is conducted to address the second conceptual question: How to intuitively understand *distributed programs* using the same conceptual model as *monolithic programs*? The study focuses on designing, implementing, and formalising a UMI framework as a Rust library. This UMI framework allows a distributed program to be migrated a monolithic program without massive changes to the syntax or structure of the original monolithic program. In addition, the semantics of the monolithic program is preserved. By formalising the core calculus of the UMI framework implemented as a distributed extension of Rust, we argue that our UMI framework extends Rust’s memory safety guarantees into a distributed setting by utilising Rust’s ownership and lifetime system in distributed memory management. In terms of addressing the conceptual question regarding to understanding the connection between monolithic programs and distributed programs, the UMI framework demonstrates that a monolithic program can be viewed as an abstraction of a distributed program. Rather than requiring programmers to implement *how* some functionalities are achieved in a distributed setting via network communication protocols and message passing, programmers only need to specify *what* these functionalities a distributed program is required to achieve in terms of a monolithic program by abstracting away the details of distributed memory management and message passing over a network.

The study discussed in chapter 4 is conducted to address the third conceptual question: How do we characterise the relationship between the *syntax* and *semantics* of programming languages? The study focuses on a thorough examination of the se-

mantics of a core calculus — System S — of a family of strategic rewriting languages that instructs syntactic transformations. More specifically, we formalise a denotational semantics and big-step operational semantics of System S, featuring errors, divergence, and non-determinism. In addition, we prove the equivalence of these two semantic models, showing that they are equally expressive, and model the same meaning of System S. We then present an axiomatic model of System S, which is a weakest precondition calculus, allowing us to reason about the executions of rewrites encoded in System S. Regarding to the conceptual question, this study demonstrate a perhaps interesting observation: As for strategic rewriting languages, the syntax and semantics are interdependent. The syntactic transformations of expressions encode the meaning for the evaluation of these expressions, and by designing and analysing three different formal models of semantics, we are able to characterise and reason about the executions of compositions of these syntactic transformations.

There is one important observation shared by all three studies: There is always a tension between the expressiveness and the elegance of abstraction when modelling programming languages. In the first study, we have observed that it is challenging to express performance related non-functional properties for containers using the declarative formal specification we have designed. In the second project, we have presented our UMI framework which provides a conceptual modelling which views a monolithic program as a functional specification of a distributed program, abstracting over the complicated network communication details while extending the memory safety guarantees provided by the monolithic program into a the distributed program. However, such an abstraction is not expressive enough to capture the failures caused by the network communication problems and server errors. In the third project, in order to formalise concise and elegant semantics of the composition of syntactic transformations, we abstract away the detail implementations of the atomic strategies and model them as partial functions. However, such an abstraction is not expressive enough to model concepts such as side effects of the execution of atomic strategies. To summarise such an observation, if the model is too detailed and precise, it may become overly specific and complicated, lose the generality, and obscure the high-level structure of the language features that it models. However, if the model is too abstract, it may lose some important aspects of the language features that it models. When we are designing a formal model to demonstrate some principled understanding of some features we care about in a programming language, we should always consider the balance between expressiveness and abstraction.

Back to the theme of this thesis — studies concerning the meaning of computer programs, we have conducted three studies relating to explore better techniques for modelling important features and components of programming languages, including container types, distributed programming, and term rewriting. These studies enable programmers to gain formal understanding of computer programs, to effectively communicate desired functionalities to computers without being overly specific, and to reason about the executions of computer programs.



Everything beautiful will eventually come to an end. Although I have been asking different questions, wondering around different paths, and searching for different angles to gain some understandings of the questions I have been asking, like the bird with the impaling thorn, everything I have been exploring eventually leads to the same direction: I am in the process of searching for the meaning of the world, especially the world of which I am the centre — and in the end the result might just be: *It is meaningless*. Nevertheless, I know, I understand, even if there is nothing there, still I search for it, till the end of my life, still I search for it.

Bibliography

- Abdullahi, S. E. and Ringwood, G. A. (1998). Garbage collecting the internet: a survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373.
- Adjukiewicz, K. (1935). Die syntaktische Konnexität. *Studia Philosophica*, 1:1–27. English translation “Syntactic Connexion” by H. Weber in McCall, S. (Ed.) *Polish Logic*, pp. 207–231, Oxford University Press, Oxford, 1967.
- Aguirre, A., Katsumata, S., and Kura, S. (2022). Weakest preconditions in fibrations. *Math. Struct. Comput. Sci.*, 32(4):472–510.
- AltSysRq (2022). Proptest: A rust property testing framework. Accessed Sep. 2022.
- Anderson, C. J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., and Walker, D. (2014). NetKAT: semantic foundations for networks. In Jagannathan, S. and Sewell, P., editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126. ACM.
- Ankerl, M. (2019). Hashmaps benchmarks – finding the fastest; memory efficient hashmap. Accessed Sep. 2022.
- Arvidsson, E., Castegren, E., Clebsch, S., Drossopoulou, S., Noble, J., Parkinson, M. J., and Wrigstad, T. (2023). Reference capabilities for flexible memory management. *Proc. ACM Program. Lang.*, 7(OOPSLA2).
- Bayer, R. (1972). Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306.
- Bierman, G. M., Gordon, A. D., Hrițcu, C., and Langworthy, D. (2010). Semantic subtyping with an smt solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP ’10*, page 105–116, New York, NY, USA. Association for Computing Machinery.

- Bonsangue, M. M. and Kok, J. N. (1992). Semantics, orderings and recursion in the weakest precondition calculus. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Semantics: Foundations and Applications, REX Workshop, Beekbergen, The Netherlands, June 1-4, 1992, Proceedings*, volume 666 of LNCS, pages 91–109. Springer.
- Boole, G. (1854). *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*, volume 2. Walton and Maberly.
- Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P., and Vittek, M. (1996). ELAN: A logical framework based on computational systems. In Meseguer, J., editor, *First International Workshop on Rewriting Logic and its Applications, RWLW 1996, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier.
- Brandt, R. (2000). *Articulating Reasons: An Introduction to Inferentialism*. Harvard University Press, Cambridge, Mass.
- Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70.
- Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O’Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO ’07*, page 185–197, USA. IEEE Computer Society.
- Cechner, P. (2014). vector vs map performance confusion. Accessed Sep. 2022.
- Chapman, S. and Routledge, C. (2009). *Ideational Theories*, pages 84–85. Edinburgh University Press, Edinburgh.
- Chen, Z., O’Connor, L., Keller, G., Klein, G., and Heiser, G. (2017). The cogent case for property-based testing. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, PLOS’17*, page 1–7, New York, NY, USA. Association for Computing Machinery.
- Chen, Z., Rizkallah, C., O’Connor, L., Susarla, P., Klein, G., Heiser, G., and Keller, G. (2022). Property-based testing: Climbing the stairway to verification. In *Pro-*

- ceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, New York, NY, USA. ACM.
- Chomsky, N. (1957). *Syntactic Structures*. De Gruyter Mouton, Berlin, Boston.
- Chomsky, N. (1975). *The Logical Structure of Linguistic Theory*. Springer.
- Chomsky, N. (2000). *New Horizons in the Study of Language and Mind*. Cambridge University Press.
- Claessen, K. and Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA. Association for Computing Machinery.
- Clebsch, S., Franco, J., Drossopoulou, S., Yang, A. M., Wrigstad, T., and Vitek, J. (2017). Orca: Gc and type system co-design for actor languages. *Proc. ACM Program. Lang.*, 1(OOPSLA).
- Cohen, E., Dahlweid, M., Hillebrand, M. A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. (2009). VCC: A practical system for verifying concurrent C. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of LNCS, pages 23–42. Springer.
- Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90.
- Cooper, E. E. and Wadler, P. (2009). The rpc calculus. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 231–242.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press.
- Costa, D. and Andrzejak, A. (2018). Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 16–26, New York, NY, USA. Association for Computing Machinery.

- Croft, W. and Cruse, D. A. (2004). *Cognitive Linguistics*. Cambridge Textbooks in Linguistics. Cambridge University Press.
- Cunha, A. and Visser, J. (2007). Transformation of structure-shy programs: Applied to xpath queries and strategic functions. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, page 11–20, New York, NY, USA. Association for Computing Machinery.
- Davidson, D. (1967). Truth and meaning. *Synthese*, 17(1):304–323.
- de Roever, W.-P. and Engelhardt, K. (1998). *Properties of Simulation*, page 73–89. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.
- de Saussure, F. (1916). *Cours de linguistique générale*. Payot, Paris.
- Dershowitz, N. (1985). Computing with rewrite systems. *Inf. Control.*, 65(2/3):122–157.
- Devesas Campos, M. and Levy, P. B. (2018). A syntactic view of computational adequacy. In Baier, C. and Dal Lago, U., editors, *Foundations of Software Science and Computation Structures*, pages 71–87, Cham. Springer International Publishing.
- Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457.
- Edouard (2020). Using c++ containers efficiently. Accessed Sep. 2022.
- Fauconnier, G. and Turner, M. (1998). Conceptual integration networks. *Cognitive Science*, 22(2):133–187.
- Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for java. In Knoop, J. and Hendren, L. J., editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM.
- Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32.
- Fredriksson, O. and Ghica, D. R. (2014). Krivine nets: a semantic foundation for distributed execution. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 349–361, New York, NY, USA. Association for Computing Machinery.

- Freeman, T. and Pfenning, F. (1991). Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 268–277, New York, NY, USA. Association for Computing Machinery.
- Frege, G. (1879). Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought [1879]. *From Frege to Gödel: A Source Book in Mathematical Logic*, 1931:1–82.
- Frege, G. (1892). On sinn and bedeutung. In Beaney, M., editor, *The Frege Reader*, pages 151–172. Blackwell.
- Fu, R., Dardha, O., and Steuwer, M. (2023). Traced types for safe strategic rewriting.
- Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., et al. (2011). Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327.
- Geeraerts, D. (2017). Lexical semantics.
- Girard, J. (1986). The system F of variable types, fifteen years later. *Theor. Comput. Sci.*, 45(2):159–192.
- Goncharov, S. and Schröder, L. (2013). A relatively complete generic hoare logic for order-enriched effects. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 273–282. IEEE Computer Society.
- Guttag, J. (1976). Abstract data types and the development of data structures. In *Proceedings of the 1976 Conference on Data: Abstraction, Definition and Structure*, page 72, New York, NY, USA. Association for Computing Machinery.
- Hagedorn, B., Lenfers, J., Koehler, T., Qin, X., Gorlatch, S., and Steuwer, M. (2020). Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.*, 4(ICFP):92:1–92:29.
- Hagedorn, B., Lenfers, J., Koehler, T., Qin, X., Gorlatch, S., and Steuwer, M. (2023). Achieving high performance the functional way: Expressing high-performance optimizations as rewrite strategies. *Commun. ACM*, 66(3):89–97.

- Hennessey, M. and Plotkin, G. D. (1979). Full abstraction for a simple parallel programming language. In Becvár, J., editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of LNCS, pages 108–120. Springer.
- Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60.
- Hinrichs, E. (1986). Temporal anaphora in discourses of english. *Linguistics and Philosophy*, 9(1):63–82.
- Hoare, C. A. R. (1969a). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.
- Hoare, T. (1969b). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.
- Hoare, T., Möller, B., Struth, G., and Wehrman, I. (2011). Concurrent kleene algebra and its foundations. *J. Log. Algebraic Methods Program.*, 80(6):266–296.
- Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. MIT Press.
- Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.
- Jackson, H. and Amvela, E. (2000). *Words, Meaning and Vocabulary: An Introduction to Modern English Lexicology*. Open linguistics series. Bloomsbury Academic.
- Johann, P., Simpson, A., and Voigtländer, J. (2010). A generic operational metatheory for algebraic effects. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 209–218. IEEE Computer Society.
- Johnson, M. (1987). *The Body in the Mind: The Bodily Basis of Meaning, Imagination, and Reason*. University of Chicago Press, Chicago.
- Jones, C. B. (1990). *Systematic Software Development Using VDM (2nd Ed.)*. Prentice-Hall, Inc., USA.
- Jung, C., Rus, S., Railing, B. P., Clark, N., and Pande, S. (2011). Brainy: Effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, page 86–97, New York, NY, USA. Association for Computing Machinery.

- Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2017a). Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL).
- Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2017b). Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL).
- Kaiser, M. and Lämmel, R. (2009). An Isabelle/HOL-based model of Stratego-like traversal strategies. In Porto, A. and López-Fraguas, F. J., editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 93–104. ACM.
- Kamp, H. (1968). *Tense Logic and the Theory of Linear Order*. PhD thesis, Ucla.
- Kamp, H. and Reyle, U. (1993). *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer Academic Publishers, Dordrecht.
- Katz, J. and Fodor, J. (1963). The structure of a semantic theory. *Language*, 39:170–210.
- Katz, J. J. (1972). *Semantic Theory*. Harper & Row, New York,.
- Kiebertz, R. B. (2001). A logic for rewriting strategies. *Electronic Notes in Theoretical Computer Science*, 58(2):138–154. STRATEGIES 2001, 4th International Workshop on Strategies in Automated Deduction - Selected Papers (in connection with IJCAR 2001).
- Klabnik, S. and Nichols, C. (2018). *The Rust Programming Language*. No Starch Press, USA.
- Koppel, J. (2023). Typed multi-language strategy combinators. In Lämmel, R., Mosses, P. D., and Steimann, F., editors, *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*, volume 109 of OASiCS, pages 16:1–16:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- Kozen, D. (1991). A completeness theorem for kleene algebras and the algebra of regular events. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 214–225. IEEE Computer Society.

- Kozen, D. (1997). Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443.
- Kozen, D. (1999). On hoare logic and kleene algebra with tests. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 167–172. IEEE Computer Society.
- Kripke, S. A. (1980). *Naming and Necessity: Lectures Given to the Princeton University Philosophy Colloquium*. Harvard University Press, Cambridge, MA.
- Lakoff, G. and Johnson, M. (2003 - 1980). *Metaphors we live by / George Lakoff and Mark Johnson*. University of Chicago Press, Chicago, [new edition with a new afterword]. edition.
- Lambek, J. (1958). The mathematics of sentence structure. *Journal of Symbolic Logic*, 65(3):154–170.
- Lämmel, R. (2003). Typed generic traversal with term rewriting strategies. *J. Log. Algebraic Methods Program.*, 54(1-2):1–64.
- Lämmel, R. (2007). Scrap your boilerplate with xpath-like combinators. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 137–142, New York, NY, USA. Association for Computing Machinery.
- Lämmel, R., Thompson, S., and Kaiser, M. (2013). Programming errors in traversal programs over structured data. *Science of Computer Programming*, 78(10):1770–1808.
- Lämmel, R. and Visser, J. (2002). Design patterns for functional strategic programming. In Fischer, B. and Visser, E., editors, *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, Pittsburgh, Pennsylvania, USA, 2002*, pages 1–14. ACM.
- Langacker, R. W. (1987). *Foundations of cognitive grammar / Ronald W. Langacker*. Stanford University Press, Stanford, Calif.
- Leino, K. R. M. (2005). Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288.

- Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In Clarke, E. M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of LNCS, pages 348–370. Springer.
- Leroy, X. and Grall, H. (2009). Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304. Special issue on Structural Operational Semantics (SOS).
- Lewis, D. (1970). General semantics. *Synthese*, 22(1/2):18–67.
- Liskov, B. and Zilles, S. (1974). Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA. Association for Computing Machinery.
- Mcgilvray, J. (1998). Meanings are syntactically individuated and found in the head. *Mind and Language*, 13(2):225–280.
- Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.
- Mitkov, R. (2022). *The Oxford Handbook of Computational Linguistics*. Oxford University Press.
- Montague, R. (1970). English as a formal language. In Visentini, B., editor, *Linguaggi nella societa e nella tecnica*, pages 188–221. Edizioni di Comunita.
- Morgan, C. (1994). *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice Hall.
- Nelson, B. J. (1981). *Remote procedure call*. PhD thesis, USA. AAI8204168.
- Newmeyer, F. J. (1986). *Linguistic theory in America / Frederick J. Newmeyer*. Academic Press, Orlando ;, second edition. edition.
- Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer.
- Orr, D. A. H. (2019). Finding unique items - hash vs sort? Accessed Sep. 2022.

- Palmer, F. (1981). *Semantics*. Cambridge low priced editions. Cambridge University Press.
- Partee, B. H. (1984). Nominal and temporal anaphora. *Linguistics and Philosophy*, 7(3):243–286.
- Pearce, D. J. (2021). A lightweight formalism for reference lifetimes and borrowing in rust. *ACM Trans. Program. Lang. Syst.*, 43(1).
- Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press, 1 edition.
- Pietroski, P. (2017). Semantic internalism. *The Cambridge companion to chomsky*, 2:196–216.
- Plotkin, G. and Power, J. (2001). Adequacy for algebraic effects. In Honsell, F. and Miculan, M., editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Plotkin, G. D. (1976). A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487.
- Portner, P. and Partee, B. H. (2002). *Formal semantics : the essential readings / edited by Paul Portner and Barbara H. Partee*. Blackwell, Oxford.
- Prior, A. N. (1955). *Time and Modality*. Greenwood Press, Westport, Conn.
- Pustejovsky, J. (1991). The syntax of event structure. *Cognition*, 41(1):47–81.
- Pustejovsky, J. (2006). Lexical semantics: Overview. In Brown, K., editor, *Encyclopedia of Language & Linguistics (Second Edition)*, pages 98–106. Elsevier, Oxford, second edition.
- Reynolds, J. C. (1974). Towards a theory of type structure. In Robinet, B. J., editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer.
- Russell, B. (1905). On denoting. *Mind*, 14(56):479–493.
- serde-rs (2023). Serde.

- Shacham, O., Vechev, M., and Yahav, E. (2009). Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 408–418, New York, NY, USA. Association for Computing Machinery.
- Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D. S., Rosenmüller, M., and Saake, G. (2012). Predicting performance via automated feature-interaction detection. In Glinz, M., Murphy, G. C., and Pezzè, M., editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 167–177. IEEE Computer Society.
- Simpson, A. (2004). Computational adequacy for recursive types in models of intuitionistic set theory. *Annals of Pure and Applied Logic*, 130(1):207–275. Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS).
- Smits, J. and Visser, E. (2020). Gradually typing strategies. In Lämmel, R., Tratt, L., and de Lara, J., editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 1–15. ACM.
- Sozeau, M. (2014). Proof-relevant rewriting strategies in coq. In *At Coq Workshop*.
- Spivey, J. M. (1989). *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., USA.
- Steedman, M. (2001). *The Syntactic Process*. The MIT Press.
- Steedman, M. and Baldridge, J. (2011). Combinatory categorial grammar. *Non-Transformational Syntax: Formal and Explicit Models of Grammar*, pages 181–224.
- Stich, S. P. and Warfield, T. A., editors (1994). *Mental Representation: A Reader*. Blackwell, Cambridge, USA.
- Stokke, B. (2022). im conslist: A rust cons-list implementation. Accessed Sep. 2022.
- Stoy, J. (1985). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Computer Science Series. MIT Press.
- Swamy, N., Hrițcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., Zinzindohoue, J.-K., and Zanella-Béguelin, S. (2016). Dependent types and multi-monadic effects in f^* . In *Proceedings*

- of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA. Association for Computing Machinery.
- Swierstra, W. and Baanen, T. (2019). A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP).
- Talmy, L. (2000). *Toward a Cognitive Semantics: Concept Structuring Systems*. The MIT Press.
- Tarski, A. (1944). The semantic conception of truth: and the foundations of semantics. *Philosophy and Phenomenological Research*, 4(3):341–376.
- Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing.
- Torlak, E. and Bodik, R. (2013). Growing solver-aided languages with rosette. In Hosking, A. L., Eugster, P. T., and Hirschfeld, R., editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM.
- Torlak, E. and Bodik, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 530–541, New York, NY, USA. Association for Computing Machinery.
- Trier, J. (1931). *Der deutsche Wortschatz im Sinnbezirk des Verstandes: die Geschichte eines Sprachlichen feldes*. Number Bd. 1 in *Der deutsche Wortschatz im Sinnbezirk des Verstandes*. Verlag nicht ermittelbar.
- Vazou, N., Rondon, P. M., and Jhala, R. (2013). Abstract refinement types. In Felleisen, M. and Gardner, P., editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer.
- Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton-Jones, S. (2014). Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Con-*

- ference on Functional Programming*, ICFP '14, page 269–282, New York, NY, USA. Association for Computing Machinery.
- Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In Middeldorp, A., editor, *Rewriting Techniques and Applications*, volume 2051 of LNCS, pages 357–361. Springer.
- Visser, E., Benaissa, Z. E., and Tolmach, A. P. (1998). Building program optimizers with rewriting strategies. In Felleisen, M., Hudak, P., and Queinnec, C., editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, pages 13–26. ACM.
- Visser, E. and Benaissa, Z. E.-A. (1998). A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441. International Workshop on Rewriting Logic and its Applications.
- Wachsmuth, G., Konat, G. D. P., and Visser, E. (2014). Language design with the spoofax language workbench. *IEEE Softw.*, 31(5):35–43.
- Wagemaker, J., Foster, N., Kappé, T., Kozen, D., Rot, J., and Silva, A. (2022). Concurrent netkat - modeling and analyzing stateful, concurrent networks. In Sergey, I., editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of LNCS, pages 575–602. Springer.
- Weiss, A., Gierczak, O., Patterson, D., and Ahmed, A. (2021). Oxide: The essence of rust.
- Wicht, B. (2012). C++ benchmark – std::vector vs std::list vs std::deque. Accessed Sep. 2022.
- Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA.
- Winter, V. L. and Subramaniam, M. (2004). The transient combinator, higher-order strategies, and the distributed data problem. *Sci. Comput. Program.*, 52:165–212.

- Wirsing, M. (1990). Algebraic specification. In Van Leeuwen, J., editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 675–788. Elsevier, Amsterdam.
- Wollrath, A., Riggs, R., and Waldo, J. (1996). A distributed object model for the javatm system. In *Proceedings of the 2nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, COOTS’96, page 17, USA. USENIX Association.
- Xu, G. (2013). Coco: Sound and adaptive replacement of java collections. In Castagna, G., editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 1–26. Springer.