This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.
Type Systems for Safe Strategic Rewriting

Rongxiao Fu

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2024
Abstract

Strategy languages enable programmers to compose rewrite rules into strategies and control their application. This is useful in programming languages, e.g., for describing program transformations compositionally, and also in automated theorem proving. Not all compositions of rewrites are correct, and it is not always easy for the programmers to spot erroneous strategy compositions in practice. Thus, we raise the question: how do we assist programmers to avoid mistakes and in writing correct composition of rewrites as strategies?

As our proposed solution, type systems are a commonly used method to rule out erroneous programs and ensure that type-checked programs have expected properties, and in this thesis, we use type systems to capture the shape of rewritten terms and reject strategies which are guaranteed to fail at runtime for all possible inputs. There are previous works in designing type systems for strategies languages, but they either focus on the interactions of generic strategies instead of the correctness of each concrete individual composition, or lack important features such as customized strategy combinators. Initially inspired by the strategy language ELEVATE, this thesis presents three statically typed strategy languages assisting programmers to write correct strategies.

The first language introduced in this thesis is Typed ELEVATE, which is equipped with a row-polymorphic type system working as the basis for the other two more advanced type systems. It uses structural row types to capture how rewriting transforms the shape of the rewritten programs, and it statically rejects strategies which lead to runtime pattern matching errors. We show a formalization of Typed ELEVATE together with its soundness proof, and the demonstration of its practical use for expressing compiler optimization strategies.

However, there are still many practically useless strategies which can normally run without runtime errors, but always return failed rewriting results. To detect such unproductive strategies, we introduce the second language in this thesis, Rewrite \( S_t \), a static type system combing the row-polymorphic types and a novel tracing system that keeps track of all possible legal strategy execution paths, while preserving as many of the useful features of Type ELEVATE as possible. Rewrite \( S_t \) introduces new concepts such as traces and unproductive strategies, but it still has many limitations such as weak formal guarantees, overly complex formalization and lack of recursive types.
After discussing the issues of Rewrite $S_t$, we introduce the third language in this thesis, Core $S_t$, which also features a traced type system but addresses most of the issues. Core $S_t$ simplifies the tracing mechanism, supports recursive types, and provides stronger formal guarantees. It raises warnings when parts of a composition are guaranteed to fail at runtime, and errors when no legal execution for a strategy is possible. We present a formalization of Core $S_t$, and formally proves its type soundness, shows that ill-traced strategies are guaranteed to fail at runtime and that well-traced strategy executions “can’t go wrong”, meaning that they are guaranteed to have a possible successful execution path.
Lay Summary

During the process of software development, the compiler translates the source code mostly written by human to instructions which can be recognized and executed by machines. To develop software with better performance, programmers normally rely on the compilers for the optimisation tasks in the hope of that the compilers can automatically perform optimisations. However, without the guidance of experts, the compilers are often not smart enough to spot the optimal routes of optimisation. To include domain-specific expertise into the optimisation process, the demand of fine and customisable control of optimisation arises.

A commonly used method for program optimisation is rule-based term rewriting, that is, transforming the program according to a set of rewrite rules to achieve higher performance. In many scenarios, it is sufficient to simply specify a set of rewrite rules which are repeatedly applied by the compiler until no rule is applicable any more. However, there are many cases where this is not adequate, and programmers expect to finely control when and where to apply which rewrite rule.

Thus, to control the application of rewrite rules, strategy languages have been proposed. Using rewrite rules as building blocks, strategy languages provide strategy combinators which compose rewrite rules to form strategies, and compose smaller strategies into larger ones. With strategy languages, programmers can specify the details of applying rewrite rules, such as the order or place of applications, and can even define their own customized strategy combinators.

However, programmers can make mistakes when writing strategies. In a problematic strategy, incompatible rewrite rules may be composed together, and the application of this strategy will always produce a failed result. Such strategies are considered to be unproductive. Thus, to prevent programmers from writing unproductive strategies, this thesis introduces three strategy languages that help to reduce errors in strategies. These languages are built on each other to detect increasingly difficult errors.
Acknowledgements

This thesis marks the successful conclusion of my journey over the past four years and eight months. This journey has been filled with both achievements and challenges, leaving me with precious memories. During this period, I have been fortunate to receive help, guidance, encouragement, and companionship from many people, and I would like to express my gratitude here.

First and foremost, I would like to express my sincere thanks to my supervisor, Prof. Dr. Michel Steuwer. Your academic expertise and enthusiasm have supported and helped me in every way. You provided essential advice and timely feedback for my research and offered comfort and assistance when I encountered difficulties. Having your guidance and companionship throughout these years of research has been both fortunate and honourable.

I am grateful to Dr. Ornela Dardha for your guidance and advice during our discussions and collaborations, as well as to Prof. James Cheney, Dr. Sam Lindley and Dr. Daniel Hillerström for your support and inspiration. I also appreciate Dr. Ohad Kammar and Dr. Jesper Cockx for taking the time to read my thesis and providing detailed and constructive feedback.

I would like to thank all the members of the research team that has gradually grown from the Lift project. The cooperation and communication within the team have broadened my horizons and provided me with many insights in my research.

I extend my gratitude to Kang Tan, Huihua Cheng, Shuzhuang Xu, Xueying Qin, and many other friends who have shared this time with me, with special thanks to Jun Ma. Your companionship and support have added countless warm and joyful moments to my journey.

Finally, I would like to thank my family, especially my parents, Zhaoxia Liao and Ji Fu. Although we have been in different countries for most of the past few years, your love has transcended the distance, becoming the driving force that has supported my progress.
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.

(Rongxiao Fu)
# Table of Contents

1 Introduction .......................... 1

2 Background ............................ 8
   2.1 Strategic Rewriting and Strategy Languages .................. 8
   2.2 Row Polymorphism .......................... 14

3 A Row-Polymorphic Type System for Strategic Rewriting ........ 17
   3.1 Introduction ............................ 17
   3.2 Typed ELEVATE by Example .................. 18
      3.2.1 Encoding Representation of Programs in Types ............ 18
      3.2.2 Strategies in Typed ELEVATE .................... 19
      3.2.3 Strategy Combinators in Typed ELEVATE ............... 22
      3.2.4 Safe Compositions in Typed ELEVATE ................. 24
   3.3 Typed ELEVATE, Formalized .................. 25
      3.3.1 Syntax .................................. 26
      3.3.2 Type System ............................ 27
      3.3.3 Operational Semantics ..................... 33
      3.3.4 Properties of the Type System .................... 33
   3.4 Implementation .......................... 36
      3.4.1 Pattern Elaboration ....................... 36
      3.4.2 Type Inference .......................... 41
   3.5 Conclusion .............................. 41

Appendix 3.A Type Soundness Proof .................. 42

4 Traces for Safe Compositions .......................... 50
   4.1 Introduction .............................. 50
   4.2 Limitation of Typed ELEVATE ................. 51
      4.2.1 Checking Sequential Compositions ................. 52
4.2.2 Checking Left-choice Compositions ........................................ 53
4.3 Rewrite \( S_t \), Intuitively ....................................................... 53
  4.3.1 Traces for Checking Strategy Compositions ......................... 53
4.4 Rewrite \( S_t \), Formalized ...................................................... 56
  4.4.1 Term Syntax ................................................................. 56
  4.4.2 Kinds and Types .......................................................... 57
  4.4.3 Type System ............................................................... 62
  4.4.4 Operational Semantics .................................................. 71
4.5 Technical Results ............................................................... 72
  4.5.1 Type Soundness ............................................................ 72
  4.5.2 On (Un)productive Strategies ........................................ 72
4.6 Discussion ...................................................................... 73
  4.6.1 Weak Formal Guarantees ............................................... 74
  4.6.2 Complex Formalization ................................................. 75
  4.6.3 Limited Flexibility ....................................................... 76
4.7 Conclusion ................................................................. 78
Appendix 4.A Formalization ....................................................... 79
  4.8.1 Typing Rules for Untraced Terms ................................... 79
  4.8.2 Type Scheme Instantiation and Type Boards ..................... 79
  4.8.3 Other Definitions and Rules ......................................... 83

5 Traced Types for Safe Strategic Rewriting ................................... 93
  5.1 Introduction ................................................................. 93
  5.2 Core \( S_t \) by Example ....................................................... 94
    5.2.1 Valid Strategy Compositions and Strategy Combinators .... 95
    5.2.2 Problematic Strategy Compositions Causing Warnings ..... 100
    5.2.3 Problematic Strategy Compositions Causing Errors ....... 101
    5.2.4 Practical Examples .................................................... 103
  5.3 Core \( S_t \), Formalized ....................................................... 106
    5.3.1 Term Syntax ............................................................. 106
    5.3.2 Type Syntax ............................................................. 108
    5.3.3 Well-formedness of Types ........................................ 114
    5.3.4 Typing Rules for Rewrite rules .................................. 117
    5.3.5 Typing Rules for strategy Combinators ....................... 120
    5.3.6 Typing Rules for Strategy Executions and let-bindings .... 129
5.3.7 Operational Semantics ............................................. 131
5.3.8 Soundness and Strong Normalization ......................... 132
5.3.9 Properties of Well-typed Strategies ............................ 135
5.4 Discussion .......................................................... 137
  5.4.1 Comparison With Rewrite $S_t$ ............................... 139
  5.4.2 Design Choices and Trade-Offs .............................. 140
5.5 Extensions ......................................................... 145
  5.5.1 Integer Pattern ................................................. 145
  5.5.2 Rewrite Condition ............................................... 147
  5.5.3 Conditional Rewrite .......................................... 148
5.6 Conclusion ......................................................... 148
Appendix 5.A Definition of trace operations ....................... 150
Appendix 5.B Proofs .................................................. 154

6 Conclusions ......................................................... 161
  6.1 A Note About Generic Traversal ................................. 163
  6.2 Related Work ...................................................... 165
  6.3 Future Work ....................................................... 171

Bibliography .......................................................... 175
Chapter 1

Introduction

The pursuit of high performance in software industry and computer science research never ends. However, in practice, high performance code often come at the price of significant changes in the source code, or even a complete reimplementation for applying optimisations. Besides, with numerous and diverse hardware in the markets, people also expect software to be performance portable, which means the relative performance of the software does not depend on any specific hardware platforms, in other words, achieving good performance on different devices. This proposes an even larger challenge for software development. Thus, developers rely on the compilers for the optimisation tasks in the hope of that the compilers can automatically locate the code with performance issues and perform suitable optimisations. However, the search space of possible optimisations is almost always extensively large, often even infinite. Without the guidance of experts, the compilers are often not smart enough to spot the optimal routes of optimisation. To include domain-specific expertise in both the application areas and the hardware platforms into the optimisation process, the demand of finely grained and customisable control of optimisation arises.

Attempts in achieving performance portability certainly exist, and some of them are prominent. Halide [Ragan-Kelley et al., 2013] is a successful and widely used Embedded Domain Specific Language (EDSL) in C++, specialised in high performance array and image processing. Halide tries to achieve performance portability by separating computations and the optimisation schedules. Programmers can write straightforward and platform-independent algorithms in Halide without worrying about the performance. The optimisation schedules applied to the algorithms, on the other hand, are written separately and target a specific platform or use case. However, the optimisation schedules are built manually by experts in Halide with fixed APIs, leav-
Chapter 1. Introduction

ing limited space for customisation, and the programmers still do not have full control of the exact behaviours of schedules.

Another commonly used method for program optimisation, which supports customisable optimisation process, is rule-based term rewriting, i.e., transforming the program according to a set of rewrite rules to achieve higher performance [Visser, 2001]. Actually, besides program transformation in compilers, rewrite systems find applications in many other domains ranging from logic [Marchiori, 1994] and theorem provers [Hsiang et al., 1992] to specifying the semantics of programming languages [Rosu and Serbanuta, 2010]. In many scenarios, it is sufficient to simply specify a set of rewrite rules—each specifying an individual rewrite step—which are applied (possibly non-deterministically) by a rewriting system until a normal form is reached or no rule is applicable any more.

However, there are many cases where this is not adequate. For example, rewrite rules are a straightforward choice for encoding simple program transformation, but are not sufficient to encode more complex program transformations and for controlling their applications in practice. It is usually required to apply a rule only to a subpart of a program, apply multiple rules in a specific order, or en-/disable rules during a specific phase. To control the application of rewrite rules, strategy languages, such as Stratego [Visser et al., 1998; Bravenboer et al., 2008] have been proposed. Kirchner [2015] provides a recent overview of the field. These strategy languages enable strategic rewriting by providing combinators to compose rewrite rules into larger strategies.

Specifically, Stratego is an integral part of the Spoofax language workbench by Kats and Visser [2010] designed to declaratively specify languages and tailored IDEs to work with them. The Stratego strategy language is used there to write interpreters and compilers purely using compositions of rewrites.

More recently, Hagedorn et al. [2020] describes the ELEVATE strategy language and how it is used to encode and control the application of traditional compiler optimizations, such as loop-tiling, compositionally. The achieved performance is comparable to the traditionally designed TVM compiler [Chen et al., 2018] for deep learning. This picks up the trend of increased importance of efficiency in many application domains of today and the future. For example, the breakthrough success of deep learning has only been possible thanks to carefully optimized software making efficient use of modern parallel hardware. In the TVM compiler, optimization decisions are encoded in Halide-style schedules where performance engineers select from a fixed set of exposed compiler transformations to optimize their deep learning application. In EL-
EVATE, strategic rewriting gives developers even greater flexibility, as they are free to encode novel program transformations—possibly domain- or hardware-specific—as strategies and precisely control their application.

In fact, ELEVATE shows how to rethink the design of “user-schedulable languages” as strategy languages, as highlighted by Ragan-Kelley [2023]. Schedules allow experts precise control over what compiler optimizations to apply, an idea popularized by Halide and now widely adopted in other optimizing compilers, such as TVM.

Closely related to strategy languages are tactic languages in computer-assisted theorem proving that allow to control the arrangement of individual proof steps. These ideas go all the way back to Gordon et al. [1978] and tactic languages are now found in many proof assistants, such as Coq [Delahaye, 2000] and Isabelle [Nipkow et al., 2002]. Even for proof assistants like Agda [Norell, 2007] where tactics are not core proof tools, the idea of rewrite rules and tactic programming still found its applications [Cockx et al., 2019; Kokke and Swierstra, 2015].

On Correct Compositions of Rewrites

Strategy languages, such as Strtego and ELEVATE, empower programmers to describe complex strategies as compositions of rewrites. However, developing strategies that encode meaningful complex program transformations such as loop tiling is not easy, and clearly, not all compositions of rewrites are correct. As a simple example, consider the sequential composition (\;) of two rewrite rules:

\[
\text{let broken} = \text{rule } x + y \rightarrow y + x; \text{rule } 1 \cdot z \rightarrow z
\]

In this composition, we first apply a commutativity rewrite, before attempting to remove the 1 as the neutral element of multiplication. Obviously, something has gone wrong here. The first rewrite will produce an expression with an addition operator, but the second rewrite expects a multiplication operator. This composition of rewrites can never work and will fail at runtime for any possible input.

In practice, such obvious problem may be easily seen, but with strategy abstractions (such as the broken here), strategies may be imported from different files and without the assistance of any analysis tools, it is often tricky to statically tell the behaviour of a strategy solely from its name. Let’s consider the following code:

\[
\text{let broken} = \text{plusComm ; multUnitL}
\]
This line of code is essentially still the example above, but the two rewrite rules are now strategy abstractions. Although they are named in an instructive way, but in practice, it is often difficult to use a single name to cover all the aspects of a possibly large strategy. Besides, for large strategies composed with smaller ones, it is also more convenient if composition errors can be statically detected and located instead of resulting in vague runtime errors.

With all these problems and needs, we raise a general question: how do we assist programmers to avoid such mistakes and in writing correct composition of rewrites as strategies?

The solution proposed by this thesis is a static type system capturing the shape of rewritten terms and rejecting strategies which are guaranteed to fail at runtime for any input. We already know that statically-typed mainstream programming languages feature types and type systems that guide developers to use compositions correctly, for example, by only allowing the composition of functions with appropriately matching types. As for the type systems described in this thesis, we use structural types to represent the shape of terms and for typechecking strategy compositions, and the precise matching of types is assisted by a novel tracing system which precisely describes the possible execution paths of strategies. The rest of this thesis will introduce the design of three closely-related strategy languages (Typed ELEVATE, Rewrite $\mathcal{S}_t$, and Core $\mathcal{S}_t$) and their increasingly more expressive type systems, and show that how these languages can help programmers write correct strategy compositions.

Contributions and Thesis Structure

After a general introduction of the thesis in this chapter, Chapter 2 will give the background of the research, including a more detailed introduction of strategic rewriting, the ELEVATE language which inspired the research work here, and the basic concept of row polymorphism which is a key component of all three languages in this thesis. Afterwards, the body of this thesis can be divided into three major parts:

- In Chapter 3, we present Typed ELEVATE, a structurally typed variant of the ELEVATE language with a row-polymorphic type system. We encode the grammar of the Abstract Syntax Tree (AST) of the rewritten terms / programs as recursive variant types. This enables the representation of computational programs at the type level. Rewrite rules in Typed ELEVATE are encoded as func-
tions whose type encapsulates the shape of the program matched by the rule as well as the program shape after applying the rewrite. Rewrite rules are implemented via pattern matching over the rewritten program’s shape. When the expected shape is matched, then the rewritten program is returned, otherwise a failure case is returned.

Because pattern matching is such a central part of implementing rewrite rules and strategies, the type system of Typed ELEVATE provides exhaustive checking of pattern matching, thus enforcing that all possible cases are covered. Our practical implementation provides a desugaring mechanism that automatically expands, convenient to write but hard to analyze, complex patterns and expand them into easier to check, simple patterns.

We formally prove that besides type soundness, the type system of Typed ELEVATE statically guarantees that rewrite strategies will not fail at runtime due to a missing case in pattern matching and do not contain dead pattern matching branches which are statically guaranteed not to be reached.

This part of work is available on arXiv [Fu et al., 2021], and have been cited in Liu et al. [2022], Liu [2022] and Chetioui et al. [2022] as an example of type systems for strategy languages.

• In Chapter 4, we present Rewrite $S_t$, which is a traced type system for strategic rewriting, preserving as many of the useful features of Typed ELEVATE as possible. Although Typed ELEVATE is capable of expressing the strategies and strategy combinators in the original ELEVATE language, there are still many unproductive strategies which definitely fail at runtime, but cannot be detected in Typed ELEVATE. Especially for strategies containing the choice combinator (\(\mid\)) that picks among two possible strategies, to statically determine whether these strategies will definitely fail at runtime is challenging or impossible for Typed ELEVATE. Thus, to address this problem, we introduce a novel tracing mechanism using special language constructs called traces in Rewrite $S_t$.

Rewrite $S_t$ inherits the row-polymorphic type-level representation of rewritten terms and adds traces to it. We give rewrite rules and their compositions as strategies function-like strategy types reflecting the syntactic program transformation described by the rewrite. To account for higher-order combinators that take strategies as arguments, we give them dedicated strategy combinator
types. In these types, we statically compute possible execution paths and encode them as traces which are basically special variables attached to the underlying structural types. For an expression executing a strategy, we firstly check if any possible execution paths or traces exist, if not, the execution is rejected because it definitely fails at runtime, otherwise we run the execution and give back the same form of rewrite result type as Typed ELEVATE.

We present the formal definition and show the properties of Rewrite \( S_t \). We discuss the limitations of Rewrite \( S_t \) in detail, such as weak formal guarantees, complex formalization and limited flexibility due to linearity and the lack of recursive types.

- In Chapter 5, we present Core \( S_t \), a language equipped with an improved type system utilizing traces, addressing most of the issues of Rewrite \( S_t \). Unlike Rewrite \( S_t \), Core \( S_t \) does not rely on existing definitions from Typed ELEVATE. As a compact and standalone language, while keeping the basic idea of tracing unchanged, Core \( S_t \) has a less verbose but more powerful tracing mechanism, a significantly simplified presentation of typing rules, and a straightforward definition of operational semantics. With the support of recursive types and more primitive strategy combinators, Core \( S_t \) allows more flexible and practical strategy definitions than Rewrite \( S_t \).

We provide the formal proof of type soundness in the presence of traces, formally define the meaning of "well-typed and well-traced", and show that strategies which are structurally well-typed and well-traced are free of composition errors and with suitable input their execution is guaranteed to have a possible successful execution path. On the contrary, we formally show that structurally well-typed strategies which are not well-traced must result in a runtime failure and, therefore, are justifiably rejected by our type system.

We also discuss the design choices and limitations in Rewrite \( S_t \), such as deterministic choice versus non-deterministic choice, robustness of formal guarantees provided by Core \( S_t \), and the lack of term-level recursions.

To design a useable type system for practical strategy programming, and provide formal guarantees which help the programmer write correctly composed strategies, the design of the strategy languages in this thesis and the corresponding design choices we made are based on three assumptions:
• **Precisely describing the shape of rewritten programs using row polymorphism.** As mentioned above, the goal of this thesis is to statically detect ill-composed strategies, where mismatched patterns are the cause of errors. To analyse pattern matchings at type-level and detect mismatched patterns, we need precise type-level information about the shapes of ASTs which are represented as or matched against patterns. In this thesis, the aforementioned type-level information is provided by structural types, specifically, row types. Row types have the ability to encode patterns and rewrite rules as types, allowing also the compositions of strategies to have precise type-level representations. Compared with other structural type systems such as structural subtyping, row polymorphism is easily compatible with the the commonly-used Hindley-Milner type inference [Damas and Milner, 1982]. This is also supported by the next assumption. More details are explained in Section 2.2.

• **Fully automated type and trace inference is required.** Strategies are composed with smaller strategies, so their structural types can quickly become massive and hard to read. Thus, manually writing type specifications for strategies is not always practical for programmers. On the other hand, the traces introduced in this thesis, represents all possible execution paths of possibly large compositions and they are entangled with the underlying row types, so it can be tricky for the programmers to manually figure out the exact traces. Supported by the next assumption, the trace information also should not be manually described because human is prone to make mistakes. Consequently, the types and traces in the languages presented in this thesis are automatically inferred.

• **Tracing information needs to be completely precise.** Traces describe the possible execution paths and serve the goal of detecting strategies with no successful execution paths, so imprecise traces easily lead to missed detections and false alarms. Besides, for one component of a strategy composition, any imprecisions in its traces propagate to other parts of the composition, resulting in imprecise description of the whole program. Thus, to accomplish the goal of detecting broken strategy compositions, the tracing information must be completely precise.

Finally, in Chapter 6, we conclude the thesis and discuss possible future works. The idea of tracing provides a way to dissect complex types and expose internal dependencies, and we are keen to see its applications in more research fields.
Chapter 2

Background

2.1 Strategic Rewriting and Strategy Languages

Program transformations as rewrite rules

Rewriting is a convenient method for encoding program transformations, particular in functional programs. For example, the Glasgow Haskell Compiler (GHC) allows defining rewrite rules in the source code with the RULES pragma so the compiler can perform transformation based on the given rules [Peyton Jones et al., 2001]. This mechanism works as the implementation of deforestation / fusion [Gill et al., 1993] in the standard library and significantly improves its performance. The implementation of a more advanced optimization technique, stream fusion, in the vector library [Coutts et al., 2007; Peyton Jones et al., 2013] also heavily uses rewrite rules.

A classic example of rewrite rules is the map fusion rule, and it can be defined in GHC as follows:

```haskell
{-# RULES "mapFusion" forall f g xs.
   map f (map g xs) = map (f . g) xs #-}
```

This definition shows the common structure of a rewrite rule: a collection of metavariables (\(f\), \(g\) and \(xs\) here), the left-hand side to be matched against the input program and the right-hand side to be instantiated based on the matching result. The rule describes a transformation replacing the composition of two consecutive list mappings (\(map\ f\) and \(map\ g\)) with a single mapping using the composition of the two element processing functions (\(f . g\)), so the intermediate data \(map\ g\ xs\) is eliminated, saving the time of storing and reading it into and from the memory. Practically, GHC uses the more general and powerful producer & consumer model [Gill et al., 1993]
for deforestation / fusion, but the major part of the implementation still relies on the RULES pragma and the term rewriting mechanism.

However, GHC performs rewriting in a straightforward but inflexible way, where all active rules will be exhaustively applied (if applicable) to the program during AST traversal. To gain some necessary yet limited flexibility, GHC provides a phase control mechanism which allows the users to configure which group of rules to apply at each simplifier phase in the compiling pipeline [Peyton Jones et al., 2001]. This is a basic method for organizing the application of rules, but it is not sufficient.

Consider the following expression: \( \text{map } f \ (\text{map } g \ (\text{map } h \ xs)) \). There could be three different rewriting results from this expression by applying the \texttt{mapFusion} rule: 
\( \text{map } (f \ . \ g) \ (\text{map } h \ xs) \), 
\( \text{map } f \ (\text{map } (g \ . \ h) \ xs) \), and 
\( \text{map } (f \ . \ g \ . \ h) \ xs \).
Depending on the context of this expression each of these results might be preferred, e.g., each result could be the starting point of subsequent rewriting steps that produce an overall better optimized program. GHC produces the last result by applying the \texttt{mapFusion} rule greedily as many times as possible. Phase control does little help here because once the \texttt{mapFusion} is active in a phase, it will be aggressively applied everywhere. For this very specific expression, there could be workaround such as defining ad-hoc rewrite rules, but in a more general sense, similar workaround cannot replace a systematic solution. The users may end up with defining, unfortunately, endless ad-hoc rewrite rules and sophisticated phase control, trying to cover all cases, and may not get expected results. This provides limitations in practice, as the simple GHC rewriting mechanism has limited its capability to perform some complex optimization tasks according to Farmer et al. [2014], where a key transformation in enhanced stream fusion cannot be expressed by the default GHC rewriting system, and a GHC plugin, HERMIT [Farmer, 2015], is used to solve this problem.

**From Rewrite Rules to Strategies**

To overcome the inflexibilities of simple rewrite systems, *strategic rewriting* is proposed: by designing a *strategy language* which supports composing individual rewrite rules into strategies, the rewriting process can be precisely controlled. ELAN [Kirchner et al., 1995; Borovanský et al., 1996; Borovanský et al., 1996] is a forerunner of this research area. Mainly designed to explore the usage of rewriting techniques for deduction and computation, ELAN gives the notion of strategies and it is able to handle both deterministic and non-deterministic applications of rewrite rules. ELAN also
inspired the design of the Stratego strategy language [Visser et al., 1998; Bravenboer et al., 2008], which has been widely applied in program analysis, transformation and synthesis. Compared with ELAN, Stratego introduces generic traversal combinators and provides more flexible rewrite rule definitions. Furthermore, Stratego also affected many other strategy languages, including a core part of the HERMIT plugin mentioned above, the Kansas University Rewrite Engine (KURE) [Sculthorpe et al., 2014], which is a strategy language embedded in Haskell, focusing on typed transformations of ASTs. More recently, Stratego has inspired the ELEVATE language that has been used to control the application of compiler optimizations encoded as rewrite rules [Hagedorn et al., 2020].

The crucial idea of ELEVATE (and Stratego) is to encode a rewrite rule as a function with a return type that enables composition in a monadic style. The map fusion rule from before is expressed in ELEVATE as follows. The syntax here is adjusted from Hagedorn et al. [2020] to match the syntax used in the rest of the thesis and to avoid confusion.

```plaintext
let mapFusion: Rise -> RewriteResult Rise =
  lam expr =
  match expr with <
  map g (map f xs) =>
  Success (map fun(x => g (f x)) xs)
  | _ => Failure
>
```

The green highlighted parts are the computational program that is rewritten expressed in the functional language RISE [Hagedorn et al., 2020]. The RISE program expr is pattern matched and if the pattern in line 4 is matched the rewritten RISE program is returned in line 5 wrapped in Success. Otherwise Failure is returned. Functions with the type Rise -> RewriteResult Rise are called strategies in ELEVATE. More generally, the rewritten language Rise in the type can be abstracted away and replaced by a type parameter if the operation is language-irrelevant. It is worth noting that this is not the only style in which strategies can be formulated. For example, instead of returning an explicit Failure when a strategy cannot be applied, the strategy language TL [Winter and Subramaniam, 2004] uses an identity-based semantics where the unchanged original term is returned in the case of failure. Besides, the well-known algebraic specification formalism ASF+SDF [Bergstra et al., 1989] also uses this identity-based style.
Strategic languages provide support for constructing more complex strategies from simpler ones. The strategy combinators [Bravenboer et al., 2008] are introduced to compose strategies and control their applications. Some commonly used strategy combinators include the sequential combinator \( ; \) which combines two strategies by applying the first one and passing its output to the second, the left-biased choice combinator \( << \) which prefers the result from applying its first strategy and only resorts to the second strategy if the first one fails, and the \texttt{try} combinator which tries to apply a strategy, but will not change the input program in the case of failure. Shown below is an exemplar definition of the \texttt{lChoice} combinator, which is \( << \) if written in the infix operator syntax.

```plaintext
let lChoice: (p -> RewriteResult p) -> (p -> RewriteResult p) ->
(p -> RewriteResult p) =

lam fs = lam ss = lam expr1 =
match (fs expr1) with <
  Success expr2 => Success expr2
| Failure => ss expr1
>
```

In this definition, the result of applying the first strategy \( fs \) on the input \( expr1 \) is matched against the \texttt{Success} and \texttt{Failure} cases, and the second strategy \( ss \) is only applied when \( fs \) fails.

Besides, many strategic languages allow defining customized strategy combinators with primitive combinators. For example, the \texttt{try} combinator can actually be defined in a self-explanatory way as follows, where the \texttt{id} rule simply returns its input.

```plaintext
let try: (p -> RewriteResult p) -> (p -> RewriteResult p) =

lam s = s << id
```

It is common for the rewritten languages to have a large amount of syntactical constructors, so specifying term traversal manually in strategies can easily end up with lots of boilerplate code and usability issues. Thus, many strategic languages provide dedicated traversal combinators to avoid manual traversal definitions [Visser, 2005]. For example, the \texttt{all} combinator applies a given strategy to all the sub-terms of the AST of a computational program, while the \texttt{one} combinator applies a given strategy to one of the applicable sub-terms. Combinators like \texttt{all} and \texttt{one} only perform one step of a complete traversal. In other words, they do not traverse further into the sub-terms. Such behaviour is called one-layer or one-step traversal. Com-
binning with recursive definitions and other combinators, we can include sub-terms in the traversal and define the topDown combinator which performs a pre-order traverse over the AST and apply the argument strategy to all encountered sub-terms, and the bottomUp combinator which performs a post-order traverse. The exemplar definitions of topDown and bottomUp in ELEVATE are as follows. These combinators are defined in Stratego similarly.

```plaintext
let topDown: (Rise -> RewriteResult Rise) -> Rise -> RewriteResult Rise = lam s = s ; all (topDown s)
let bottomUp: (Rise -> RewriteResult Rise) -> Rise -> RewriteResult Rise = lam s = all (bottomUp s) ; s
```

In languages like ELEVATE and ELAN [Borovansky et al., 2001], these traversal combinators are defined in an ad-hoc or specific manner where for each type of AST node containing sub-terms, the way of visiting its sub-terms needs to be specified. For example, in the implementation of ELEVATE, to perform the (all s) traversal on the function application AST node App(f, e) in Rise, we need to specify that the sub-terms f and e are extracted via pattern matching, and then the argument strategy s is applied on f and e in sequence. This traversal only succeeds when both applications of s to f and e succeed, and then the rewritten results can be collected in a newly constructed App node.

Meanwhile, other languages such as Stratego provide more flexible generic traversal combinators which can uniformly apply argument strategies to the sub-terms regardless of the parent AST node. In other words, the generic traversal combinators can be used on arbitrary AST node without defining the traversal process above as in ELEVATE. Concretely, although ELEVATE and Stratego have similar definition of topDown, ELEVATE requires an all combinator specifically defined for the rewritten language (RISE for example), while Stratego uses a generic all which works for arbitrary rewritten language. As a result, the topDown combinator in ELEVATE can only be applied on RISE terms, while the topDown in Stratego stays generic and can be used for any rewritten language.

The support of generic traversal is not the focus of this thesis, but is still feasible as a future extension. More discussion about this can be found in Section 5.4.2 and Section 6.1
Types for Strategy Languages

It is easy to make mistakes when writing optimization strategies in strategy languages such as ELEVATE. We know that static types are a good tool to assist the programmer in writing meaningful programs by avoiding errors that can be spotted by the compiler statically without running the program. Unfortunately, while ELEVATE is a functional language with a basic type system, e.g. distinguishing function types from the type of the computational program AST, its type system is of limited use as all strategies share the same function type. This is also the case for many other strategy languages.

In ELEVATE, programmers can tell little information about the properties of a strategy from its type. This information loss can already be seen from the \texttt{mapFusion} example above because \texttt{mapFusion} does not produce useful result for all valid RISE expressions—it only successes when the input expression is two consecutive list mappings. Furthermore, the sequential composition (\texttt{mapFusion ; mapFusion}) would require the input expression to be three consecutive list mappings, but it still has type \texttt{Strategy Rise}, making no difference with a single \texttt{mapFusion} rule. Similarly, by design it is obvious that the \texttt{try} combinator will never fail, but this is not reflected in its type. Therefore, all combinations of strategies are allowed to compose in ELEVATE as all strategies have the same function type, but there are clearly cases of compositions that will always result in failure, such as the example in Chapter 1 which is a sequential composition where the first rewrite rule produces a program shape that the second rewrite rule never matches. Thus, to address these problems and provide precise typing for strategies, in this thesis, we present several strategy languages inspired by ELEVATE, but with more advanced type systems. In these type systems, we use structural types to capture the shape of programs and later introduce traces to precisely describe the behaviour of strategies.

As for other type systems for strategy languages, Lämmel and Visser [2002] and later Lämmel [2003] focused on the typing of generic strategies which often contain generic traversal combinators, and developed a static type system for Stratego that broadly categorizes rewritten expressions into different “sorts” and bridges the gap between relatively common many-sorted strategies and generic strategies. Their work focuses on the interactions of strategies and encapsulates the shape of rewritten expressions with nominal sorts, so it is unable to detect the fine-grained incompatibilities in the rewritten expressions. On the other hand, our work focuses on a different
aspect of strategic rewriting, that is, the correctness of each individual composition. We prevent strategies from applying to terms of mismatched shapes or being composed in unproductive ways (which are guaranteed to fail at runtime).

Mametjanov [2010] and Mametjanov et al. [2011] show a solution to detect the failed composition in Chapter 1. Their work presents a structural type system capturing the shape of rewritten programs and collect all possible execution paths of strategies with union types. By applying corresponding analysis, their type system can statically detect errors such as inaccessible branches in left-biased choice compositions and sequential compositions which always fail at runtime. However, their type system only supports ad-hoc analysis for a limited number of strategy combinators, and it does not provide any way for the users to define customized strategy combinators. On the other hand, addressing very similar problems, our work provides a more general and systematic solution with stronger formal guarantees.

More recently, Smits and Visser [2020] shows the type system of Core Gradual Stratego which use gradual type system to combine the flexibility of dynamic typing and the safety provided by static typing. Koppel [2023] uses the Compositional Data Types Haskell library, which is initially designed to address the expression problem and used for generic programming [Bahr and Hvitved, 2011], to ensure the existence of applicable language constructs during rewriting and achieve type-preserving program transformations. Like Lämmel [2003] mentioned above, these researches treat the type of rewritten programs as a broad description or encapsulation, while our work looks into the concrete shapes of rewritten programs and examines the correctness of each individual composition.

### 2.2 Row Polymorphism

As mentioned above, to provide precise typing for strategies, structural types are used to encode the expected shape of rewritten programs in the type of strategies. In practical usage, there are many scenarios where a strategy dealing with multiple kinds of expressions is passed one case of them, indicating the presence of some sort of subtyping relation. Indeed, structural subtyping can be used to describe ASTs in types, but introducing the subtyping relation will increase the difficulty and complexity of type inference. The existing strategies in ELEVATE can already be enormous, not mentioning the further composition of them, so the lack of fully fledged type inference could potentially add undesirable coding burden and raise usability problems.
Thus, another form of polymorphism which is capable of expressing AST structures in types and works well with the commonly-used Hindley-Milner type inference [Milner, 1978; Damas and Milner, 1982], row-polymorphism [Wand, 1987], is chosen as the base of all type systems presented in this thesis. Row polymorphism has similar flexibility and expressiveness as structural subtyping. A recent and detailed comparison of their expressiveness can be found in Tang et al. [2023] where the author of this thesis is involved. The most well-known practical application of row polymorphism is the support of polymorphic variants and objects (polymorphic records) in the OCaml language [Garrigue, 2000, 2001].

Firstly introduced by Wand [1987], row polymorphism is a form of parametric polymorphism which allows to represent an extensible structure in types as a row, usually an inductively constructed sequence of label-type pairs, which can end with either an empty row or a row variable, and the instantiations of the row variable extend the row. Rows are originally used as the basis for constructing polymorphic variants and records, but they are also used to represent effect types in the implementation of algebraic effects and effect handlers [Hillerström and Lindley, 2016]. For the subsequent development of row polymorphism after Wand [1987], Harper and Pierce [1990] introduced the lacks and has constraints for the row variables, describing the presence and absence of specific labels in the instantiations of the constrained row variables, but later Gaster [1998] showed that only lacks constraint would be sufficient, and this conclusion supported the design of many row polymorphic type systems. Rémy [1989] introduces the presence and absence flags for labels and the corresponding presence polymorphism. Later in Rémy [1994], row polymorphism via row variables and presence polymorphism are combined. In Gaster and Jones [1996] and Gaster [1998], Harper and Pierce style row polymorphism is formulated as an application of the qualified types framework [Jones, 2003], while Pottier and Rémy [2004a] present a Rémy style row polymorphic type system and its type inference within the $HM(X)$ framework. Blume et al. [2006] presented the language $MLPolyR$ which uses rows to realize polymorphic variants and records, and $MLPolyR$ significantly influenced the design of row polymorphic types in this thesis. $MLPolyR$ treats cases or pattern matching branches as first-class values and allows pattern matching expressions to be extended with new cases at any moment. However, despite having similar formalization of rows, as a design choice, Typed ELEVATE in Chapter 3 does not support this kind of extension to get convenient exhaustiveness checking. Fortunately, as Blume et al. [2006] point out, this design choice can have the same exten-
sibility as their language if the typing rule for pattern matching reduces / refines the type of the matched term case by case, which is exactly what Typed ELEVATE does. More recently, Morris and McKinna [2019] introduce a general theory for existing row polymorphic type systems, which focuses on row concatenations and gives rise to the language ROSE as a flexible tool for programming with extensible data types.
Chapter 3

A Row-Polymorphic Type System for Strategic Rewriting

3.1 Introduction

In this chapter, we present Typed ELEVATE, a structurally typed variant of the ELEVATE language with a row-polymorphic type system. It is also used as the basis for the other two type systems presented in this thesis. In Typed ELEVATE, rewrite rules are implemented via pattern matching over the rewritten programs, whose shapes are encoded as row-polymorphic variant types. This enables Typed ELEVATE to statically reject strategies which lead to runtime pattern matching errors.

In this chapter we present the following:

- **Row-polymorphic type system**: We present a row-polymorphic type system for strategy languages that integrates exhaustive checking of pattern matching (Section 3.3).

- **Properties of type system**: We show that our type system is sound and is able to statically prevent errors such as missing cases in pattern matching and accessing non-existent fields in records (discussion at the end of Section 3.3).

- **Implementation of the type system**: We present a practical implementation of our type system including type inference and pattern elaboration (Section 3.4).

- **Program transformation case study**: We present detailed examples and a discussion of the practical use and benefits of the type system for strategic rewriting of program transformations (Section 3.2).
3.2 Typed ELEVATE by Example

In this section we give an overview of our type system and its benefits for strategic rewriting. The type system is formally introduced in Section 3.3 and its implementation discussed in Section 3.4. Our type system is designed for the ELEVATE language, however it is not limited to only ELEVATE and can directly be applied to other strategy languages. In this section we will show examples inspired from using ELEVATE for optimizing computational programs expressed in the functional language RISE as discussed in Hagedorn et al. [2020].

We will introduce a formal syntax of our type system in Section 3.3. For this section we will use standard functional programming notation and point out syntactic constructs as we go along.

3.2.1 Encoding Representation of Programs in Types

To enable the type system to reason about strategies transforming programs we must encode a representation of computational programs as types.

We encode the AST of the functional computational language RISE as follows:

```plaintext
type Rise = t as <
  Id: {Name: Nat | *}
  | Lam: {Param: Nat | Body: t | *}
  | App: {Fun: t | Arg: t | *}
  | Primitive: <Map: {*} | Zip: {*} | Reduce: {*} | *>
  | *
>
This is a recursive row-polymorphic variant type written with angle brackets <> that enclose a row - a list of label-type-pairs whose elements are separated by a vertical bar | and terminated by the empty row *. The rows here are closed because they cannot be extended. The variant type lists all possible syntactic options for a typical functional language: Identifier, Lambda expressions, and function Application. Additionally, RISE provides a set of built-in Primitives for expressing data-parallel computations.

Record types are written with curly braces { } enclosing a row where each label corresponds to a record field such as the Name of an identifier. Lambdas have two record fields: for the Parameter and the Body; and function applications have also two fields: the Function to call and the Argument.
In the definition, the type variable \( t \) enables to refer to the overall type, matching the recursive definition of an AST.

RISE expressions such as \( \text{map } g \ (\text{map } f \ xs) \) can be desugared into \( \text{app}(\text{app} (\text{map} , g), \text{app} (\text{app} (\text{map} , f), xs)) \) and then be represented in Typed ELEVATE with a matching type as follows:

```
-- expression:
App \{\text{Fun}: \text{App} \{\text{Fun}: \text{Primitive Map } | \text{Arg}: g \},\!
      \text{Arg}: \text{App} \{\text{Fun}: \text{App} \{\text{Fun}: \text{Primitive Map } | \text{Arg}: f \},\!
                          \text{Arg}: xs\}\}

-- type:
: <App: \{\text{Fun}: <App: \{
    \text{Fun}: <\text{Primitive}: <\text{Map}: \{\ast\} | > | > | > |
    \text{Arg}: g | \ast\} | > | > |
    \text{Arg}: <App: \{
        \text{Fun}: <App: \{
            \text{Fun}: <\text{Primitive}: <\text{Map}: \{\ast\} | > | > | > |
            \text{Arg}: f | \ast\} | > | > |
            \text{Arg}: xs | \ast\} | > | \ast\} | > | > |
```

As shown above, besides the empty row *, some rows terminated with spaces. These spaces are implicit type variables for row extensions, which are omitted for simplicity, and they will be explained in more detail in the next section. Correspondingly, such rows are open because they can be extended.

### 3.2.2 Strategies in Typed ELEVATE

Strategies are encoded as functions. In original ELEVATE a strategy has the type: \( \text{Rise } \rightarrow \text{RewriteResult Rise} \). In our row-polymorphic Typed ELEVATE, strategies have a more precise type allowing to capture the shape of the input program \( p_1 \) and the rewritten program \( p_2 \):

```
type \text{Strategy} = \forall p_1 \ p_2. \ p_1 \rightarrow \text{RewriteResult} \ p_2
```

where \( \text{RewriteResult} \) is the rewritten program or failure:

```
type \text{RewriteResult} = \forall p. \ < \text{Success}: p | \text{Failure}: \{\ast\} | \ast> >
```

The \( \text{RewriteResult} \) defined in Hagedorn et al. [2020] contains the failed strategy in the \text{Failure} case. This is handy when all the transformed programs have the same type, but when we have more detailed types in Typed ELEVATE, the definition in
Hagedorn et al. [2020] will lead to unwanted restrictions and increase the difficulty of coding.

**MapFusion in typed ELEVATE**

Listing 3.1 shows the mapFusion rewrite rule that we have seen earlier in Section 2.1 implemented in our row-polymorphically typed ELEVATE.

```plaintext
1 let mapFusion = lam expr = match expr with <
2    -- map g (map f xs)
3    App {Fun: App {Fun: Primitive Map | Arg: g} |
4           Arg: App {Fun: App {Fun: Primitive Map |
5              Arg: f} |
6                  Arg: xs}} =>
7    -- Success ( map fun(x => g (f x)) xs )
8    Success (App {Fun: App {Fun: Primitive Map |
9               Arg: Lam { Param: 0 | Body: App {Fun: g |
10              Arg: App {Fun: f | Arg: Id {Name: 0}}} | |
11                 Arg: xs}}>)
```

Listing 3.1: Implementation of the mapFusion rule as a strategy in Typed ELEVATE

The green highlighted parts in line 3–6 are the pattern and in lines 8–10 the rewritten expression, where variable names in lambda expressions are encoded with distinct natural numbers (0 here).

This implementation is similar to the implementation in the original weakly typed ELEVATE language with one important difference: before we had to provide a default case for when the input program did not have the expected shape; now we rely on the more advanced type system to ensure that this strategy is only applicable when the input program has the expected shape. This is also reflected by the inferred type in Listing 3.2 capturing the behaviour of the strategy statically.

```plaintext
1 <App: {Fun: <App: {
2       Fun: <Primitive: <Map: {*} | *> | *> | |
3          Arg: g | } | *> | |
4         Arg: <App: {
5            Fun: <App: {
6               Fun: <Primitive: <Map: {*} | *> | *> | |
7                 Arg: f | } | *> | |
8                Arg: xs | | *> | | } | *
9                ->
```
With syntactic sugar we might read this type as:

\[
\langle \text{map } g (\text{map } f \; xs) \mid * \rangle \rightarrow \langle \text{Success } (\text{map } \text{fun}(x \Rightarrow g(f(x))) \; xs) \mid > \rangle
\]

The argument type in line 1–8 describes the shape of the input program corresponding to the pattern from Listing 3.1. The return type describes the rewritten program shape in lines 10–19. The variables \(g\), \(f\), and \(xs\) in the pattern of Listing 3.1 correspond directly to the type variables in Listing 3.2.

**Compatibility of types**

As mentioned in Section 3.2.1, not all rows in variant types are terminated by the empty row \(*\). For a simpler example, we consider the type \(\langle \text{Failure: } \{\ast\} \mid > \rangle\) we infer for the expression \(\text{Failure}\). For cases such as this when the final entry in a row is omitted this indicates the presence of an implicit type variable, so that the type is equivalent to \(\langle \text{Failure: } \{\ast\} \mid t > \rangle\). This implicit type variable makes the type compatible with other types. For example, we want to be able to pass the \(\text{Failure}\) value to a function that expects a \(\text{RewriteResult}\) as argument. To make the argument and parameter type compatible we simply instantiate the implicit type variable with the \(\text{Success: } p \mid *\) row, which would not be possible without the implicit type variable. Instantiation of row variables that ensures the well formedness of rows is formalized in Section 3.3.
Exhaustive Checking and Pattern Elaboration

Pattern matching is a central aspect of our language as it is integral for implementing strategies such as seen in Listing 3.1. An important guarantee that we want to provide is that a strategy does not fail at runtime due to a missing case in the pattern matching. We also want to warn developers if they provide a case for which we can statically conclude that it is impossible to be reached.

Section 3.3 details the typing rules that provide these guarantees. To simplify our formal system we reduce the complexity of patterns that we need to formally reason about. More specifically, we only consider simple patterns that are either a variable pattern (x), a unit pattern ({*}), or a label l followed by a variable (l x). But the pattern in Listing 3.1 is clearly neither of these three simple cases. How do we deal with such more complex patterns?

Section 3.4 discusses a pattern elaboration mechanism that rewrites complex patterns into a sequence of simple patterns that our type system is able to check for exhaustiveness. Listing 3.3 shows the implementation of the mapFusion strategy after pattern elaboration. Variables introduced by pattern elaboration are prefixed with #. The single match expression has been replaced with a series of nested match expressions, each with a simple label or variable pattern. The record pattern in Listing 3.1 is now decomposed into individual accesses to the record fields. The typing rules presented in Section 3.3 guarantee that this expression will not fail at runtime due to a pattern matching failure.

3.2.3 Strategy Combinators in Typed ELEVATE

Strategy languages provide combinators for composing simple strategies into more complicated strategies. Listing 3.4 shows the implementation of some key combinators and their inferred row-polymorphic types in ELEVATE.

The id and fail strategies are not combinators but useful building blocks for larger compositions. The inferred types are as expected reflecting at the type level the runtime behaviour of the strategy.

The sequential combinator first applies strategy fs (line 10) before inspecting the result and applying the second strategy ss in line 11 only if the first strategy has been successful. The inferred type reflects this behaviour: the successful rewritten program p2 in the type of the first strategy (line 7) must be the input program of the second strategy (line 8). The composed strategy has a type (line 9) combining the
input type of the first strategy (p1) with the return type of the second. Here the row variable r can either represent the Success case with the rewritten program, or it can be instantiated with the empty row type when the second strategy is guaranteed to result in Failure (as the fail strategy does). The notation here indicates that the row variable r can be instantiated with any label except Failure. The kinding system for row variables will be explained in detail in Section 3.3.

The lChoice combinator prefers the successful outcome of the first strategy and only applies the second strategy if the first fails. The inferred type is interesting, as it seems that the two strategies as well as the combined strategy all return the same rewritten program p2 on success (lines 15–17). This is because in this type system it is not statically possible to determine which successfully rewritten program will be chosen at runtime and, therefore, the type system unifies both possible outcomes. We will introduce more expressive type systems in Chapters 4 and 5 which overcome this imprecision.

Finally, the try combinator applies the given strategy and in the failure case reverts back to unchanged input program. The implementation uses lChoice and id and we can observe from the inferred type that this combinator can never fail due to the absence of Failure in the return type. This type can be easily derived from the type of lChoice with the row variable r instantiated with the empty row as id can never result in Failure.
Chapter 3. A Row-Polymorphic Type System for Strategic Rewriting

Listing 3.4: Strategy combinators and their types in ELEVATE.

3.2.4 Safe Compositions in Typed ELEVATE

Using the combinators we can now compose strategies. For example, sequentially composing the mapFusion strategy twice: mapFusion ; mapFusion results in the following type that we show only using syntactic sugar for brevity:
From the type we observe that the input program of this program transformation must be a sequence of three applications of the `map` function and that the rewritten program contains only a single instance of `map`.

But what happens when we try to compose strategies that are impossible to compose? For example, consider the `reduceMapFusion` rule from Hagedorn et al. [2020]:

```plaintext
reduce g init (map f xs) =
    reduce fun(a => fun(x => g a (f x))) init xs
```

Encoding this rule as an ELEVATE strategy and trying to sequentially compose this rule with the `mapFusion` strategy leads to a type error. This is because the type system cannot find an instantiation of the type and row variables that would make the two types compatible and the composition safe.

We have seen in this section the practical use of an advanced type system for a strategy language encoding program transformations. Our type system provides important guarantees ensuring the safe composition of strategies as well as checking the exhaustiveness of pattern matching guaranteeing that pattern matching cannot fail at runtime. Next, we give a formal account of our type system.

### 3.3 Typed ELEVATE, Formalized

In this section, we present a core calculus for ELEVATE. We show the syntax of terms and types and the type system leveraging row-polymorphism.

For the formalization in this thesis, unless otherwise stated, elements in `NavyBlue` are meta-level descriptions, for e.g., the square brackets `[ ]` indicate option in the EBNF grammar; the indexed multiple occurrences (possibly separated by either | or ,) of a syntactical construct `expr` are collectively represented by `(expr)_{i \in \mathbb{N}}`, where the index (written as i, j, k, p or q) ranges over a possibly empty subset (written as M, N, U, V, etc.) of the set of natural numbers. When two index sets occur in the same formula, they are disjoint by default.
Terms

\[ e := x, y, z, \ldots | e_1 e_2 | \lambda x = e | \text{let } f = e_1 \text{ in } e_2 | \text{fix} \]

\[ l e \mid \{(l_i : e_i)_{i \in \mathbb{N}}\} \mid e.L \mid e.-l \mid e.\{+\}(l_i : e_i)_{i \in \mathbb{N}} \]

\[ \text{match } e \text{ with } \langle \cdot \Rightarrow e_1 \rangle \mid \text{match } e \text{ with } \langle l x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle \]

Figure 3.1: Syntax of terms

3.3.1 Syntax

Terms

Figure 3.1 shows the syntax of terms and patterns. Terms (denoted by \( e \)) include common constructs such as variables ranging over \( x, y, z, \ldots \), term applications, lambda abstractions, let-bindings, and the fixed point combinator. In addition, terms include the following new constructs: label applications (denoted by \( l e \)) for constructing variant values, record constructors (denoted by \( \{(l_i : e_i)_{i \in \mathbb{N}}\} \)), field accesses (denoted by \( e.L \)), field removals (denoted by \( e.-l \)), record modifications (denoted by \( e.\{+\}(l_i : e_i)_{i \in \mathbb{N}} \)), where the order of label-term pairs is insignificant and labels are all different; and finally pattern matchings: a term \( e \) can be matched with the empty pattern (\( \text{match } e \text{ with } \langle \rangle \)), the unit (empty record) pattern (\( \text{match } e \text{ with } \langle \cdot \Rightarrow e_1 \rangle \)) or the variant pattern (\( \text{match } e \text{ with } \langle l x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle \)) which introduces a variable \( x_1 \) for the case of label \( l \), and a variable \( x_2 \) representing the rest of the cases. In the rest of this paper, we may omit the \( x_1 \) for simplicity if it is immediately matched against the empty record, that is, \( \text{match } e \text{ with } \langle l \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle \) means \( \text{match } e \text{ with } \langle l x_1 \Rightarrow \text{match } x_1 \text{ with } \langle \cdot \Rightarrow e_1 \rangle \mid x_2 \Rightarrow e_2 \rangle \)

Kinds and types

Figure 3.2 shows the syntax of kinds and types in Typed ELEVATE, whose design is influenced by Blume et al. [2006]. Types are of two kinds: \( T \) – for ordinary types, and \( R \) – for row types. Ordinary types (denoted by \( t \)) include type variables (denoted by \( \alpha \)), type constructor applications or contractive types [MacQueen et al., 1984] (denoted by \( \tau \)), and equi-recursive types (denoted by \( \alpha \text{ as } \tau \)). We require types appearing under an equi-recursive binder to be contractive, which excludes meaningless types such as \( \alpha \text{ as } \alpha \) and guarantees the existence of an unique solution to the recursive equation(s) [Rémy, 2020; Im et al., 2013]. Contractive types (denoted by \( \tau \)) include function types (denoted by \( t_1 \to t_2 \)), record types (denoted by \( \{\rho\} \)) and variant types (denoted by \( \langle \rho \rangle \)).
Row types (denoted by \( \rho \)) are sequences of label-term pairs \( l : t \) ending with row variables or empty rows (denoted by \( \cdot \)), where the order of the label-term pairs is insignificant, and labels are all distinct. Rows are differentiated from ordinary types by their kinds, row kinds (denoted by \( R \)), which can be positive (denoted by \( L \), a finite subset of the set of all labels) or negative (denoted by \( \neg L \), a cofinite subset of the set of all labels) descriptions of sets of labels. The set of all possible labels and the set of all possible variable names are disjoint. Unlike Blume et al. [2006] where a row kind is associated with a row variable and describes the finite set of labels that the row variable must not contain, kind of a row in Typed ELEVATE represents the possibly infinite/cofinite set of labels which can appear in this row. For e.g., the kind of the empty row is \( \{} \), indicating that no label can appear in the empty row; given a row variable \( r \) of kind \( \neg \{A, B\} \), the row \(( A : a \mid r )\) has kind \( \neg \{B\} \), which means any label except \( B \) can appear in the row (cf. detailed explanation in Section 3.3.2).

Type schemes (denoted by \( \sigma \)) represent possibly universally quantified types. The kind of the bound type variable should be specified at the binding site.

### 3.3.2 Type System

Before the type system for Typed ELEVATE, we give in Figure 3.3 the kinding (denoted by \( \Delta \)) and typing environments (denoted by \( \Gamma \)): they can be either empty (denoted by \( \cdot \)) or extended with a type variable and its kind (denoted by \( \alpha : \kappa \)) or a variable and its
As mentioned in Section 3.3.1, the kind of a row represents the set of labels that can appear in this row, and the set can be described positively or negatively. When only negative row kinds, i.e., the set of labels that the row must not contain, are used, this kinding mechanism is similar to the lack relation in Blume et al. [2006] or the row kind in Pottier and Rémy [2004b], disallowing the construction of ill-formed rows like $ (A : \alpha \mid A : \alpha \mid r) $. With positive row kinds, more meaningful restrictions can be added to rows.

For example, given a row variable $ r $ of kind $ \neg \{True\} $, meaning (the substitution for) $ r $ must not contain the label $ True $, the type $ \langle True : \{\cdot\} \mid r \rangle $ can be unified with the whimsical type $ \langle True : \{\cdot\} \mid Apple : \{\cdot\} \mid \cdot \rangle $ because the kind of $ \langle Apple : \{\cdot\} \mid \cdot \rangle $ is $ \{Apple\} $, meaning the label $ Apple $ can appear in this row, and it is compatible with $ \neg \{True\} $. However, if the kind $ \{False\} $ is initially assigned to $ r $, the unification above is impossible and $ \langle True : \{\cdot\} \mid r \rangle $ can only be unified with more sensible types like $ \langle True : \{\cdot\} \mid False : \{\cdot\} \mid \cdot \rangle $. Although this kinding mechanism does not stop users from getting seemingly useless types like $ \langle True : \{\cdot\} \mid False : \{\cdot\} \mid \cdot \rangle $ where the label $ False $ is paired with the empty variant type $ \langle \cdot \rangle $, it allows more precise specification of the ranges of labels than solely negative row kinds.

Figure 3.4 gives the computation rules for row kinds, defining the row kind subset relation (denoted by $ \sqsubseteq $) and the row kind extension operator (denoted by $ + $). The row kind subset relation is essentially defined as the subset relation of the sets represented by the row kinds, and the row kind extension operator is defined as element insertion without duplicates.

Figure 3.5 gives the kinding rules, which also work as the well-formedness rules for types. Kinding judgments are of the form $ \Delta \vdash t : \kappa $, stating that the type $ t $ has kind $ \kappa $ (and is well formed) in the kinding environment $ \Delta $. Most of the kinding rules are straightforward. The most important rule among them is $ K-RowExtension $ showing the usage of the row kind extension operator: it extends the kind with a label and rules out rows containing repeated labels.
\[
\begin{align*}
\{l_i \mid i \in \mathcal{N}\} & \subseteq \{l_j \mid j \in \mathcal{M}\} & \text{R-Subset-pp} \\
\{(l_i)_{i \in \mathcal{N}}\} & \subseteq \{(l_j)_{j \in \mathcal{M}}\}
\end{align*}
\]

\[
\begin{align*}
\{l_j \mid j \in \mathcal{M}\} & \subseteq \{l_i \mid i \in \mathcal{N}\} & \text{R-Subset-nn} \\
\neg\{(l_i)_{i \in \mathcal{N}}\} & \subseteq \neg\{(l_j)_{j \in \mathcal{M}}\}
\end{align*}
\]

\[
\begin{align*}
\{l_i \mid i \in \mathcal{N}\} \cap \{l_j \mid j \in \mathcal{M}\} & = \emptyset & \text{R-Subset-pn} \\
\{(l_i)_{i \in \mathcal{N}}\} & \subseteq \neg\{(l_j)_{j \in \mathcal{M}}\}
\end{align*}
\]

\[
\begin{align*}
l & \notin \{l_i \mid i \in \mathcal{N}\} \\
\{l\} \cup \{l_i \mid i \in \mathcal{N}\} & = L & \text{R-Ext-p} \\
\{(l_i)_{i \in \mathcal{N}}\} + l & = L & \text{R-Ext-1}
\end{align*}
\]

Figure 3.4: Computation rules for row kinds

\[
\begin{align*}
\alpha : \kappa & \in \Delta \\
\Delta + \alpha : \kappa & \text{ K-Var} \\
\Delta + t_1 : \mathcal{T} & \Delta + t_2 : \mathcal{T} & \text{ K-FunctionType} \\
\Delta + t_1 \rightarrow t_2 : \mathcal{T}
\end{align*}
\]

\[
\begin{align*}
\Delta + \rho : \mathcal{R} & \text{ K-RecordType} \\
\Delta + \{\rho\} : \mathcal{T}
\end{align*}
\]

\[
\begin{align*}
\Delta + \rho : \mathcal{R} & \text{ K-VariantType} \\
\Delta, \alpha : \mathcal{T} + \tau : \mathcal{T} & \Delta + \alpha \text{ as } \tau : \mathcal{T} & \text{ K-RecursiveType}
\end{align*}
\]

\[
\begin{align*}
\Delta + : \{\} & \text{ K-EmptyRow} \\
\Delta + \rho : \mathcal{R} & \Delta + t : \mathcal{T} & \mathcal{R} + l = \mathcal{R}_{ext} & \text{ K-RowExtension} \\
\Delta + (l : t | \rho) : \mathcal{R}_{ext}
\end{align*}
\]

Figure 3.5: Kinding rules
Chapter 3. A Row-Polymorphic Type System for Strategic Rewriting

\[
\begin{align*}
x : \sigma & \in \Gamma \quad \Delta \vdash \sigma \leq t & \quad \text{T-Var} \\
\Delta; \Gamma \vdash x : t & & \\
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma \vdash f : t_1 \rightarrow t_2 \quad \Delta; \Gamma \vdash e : t_1 & \quad \text{T-App} \\
\Delta; \Gamma \vdash f \ e : t_2 & \\
\end{align*}
\]

\[
\begin{align*}
(\alpha_i)_{i \in N} = ftv(t_1) \setminus ftv(\Gamma) \\
\Delta, (\alpha : \kappa_i)_{i \in N}; \Gamma \vdash e_1 : t_1 \\
\sigma = (\forall (\alpha : \kappa_i)_{i \in N} t_1 \\
\Delta; \Gamma \vdash e : t, \sigma \vdash e_2 : t_2 & \quad \text{T-Inst-Row} \\
\Delta; \Gamma \vdash \text{let } f = e_1 \text{ in } e_2 : t_2 & \\
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma \vdash \lambda x = e : t_x \rightarrow t_e & \quad \text{T-Lam} \\
\Delta; \Gamma \vdash \lambda x = e : t_x \rightarrow t_e & \\
\Delta; \Gamma \vdash \text{fix} : ((t_1 \rightarrow t_2) \rightarrow t_1 \rightarrow t_2) \rightarrow t_1 \rightarrow t_2 & \quad \text{T-Fix} \\
\Delta; \Gamma \vdash e : t & \quad \Delta \vdash \rho : \mathcal{R} \quad \mathcal{R} \subseteq \neg\{l\} & \quad \text{T-Label} \\
\Delta; \Gamma \vdash \langle l : t \mid \rho \rangle & \\
\end{align*}
\]

Figure 3.6: Typing rules for basic terms

From Figure 3.6 to Figure 3.10, we presents the typing rules for terms in Typed ELEVATE, with typing judgments of the form \( \Delta; \Gamma \vdash e : t \), stating that the term \( e \) has type \( t \) in the kinding environment \( \Delta \) and typing environment \( \Gamma \), and we always assume but omit for simplicity that \( t \) and all the types in \( \Gamma \) are well-kinds in \( \Delta \). The notation \( ftv(t) \) or \( ftv(\Gamma) \) stand for the set of free type variables in type \( t \) or typing environment \( \Gamma \), respectively. The typing rules for the lambda-calculus subset of Typed ELEVATE (T-Var, T-App, and T-Lam) are standard. The type scheme instantiation relation \( \leq \) used by the T-Var rule is defined in Figure 3.7, where substituting type \( t \) for type variable \( \alpha \) in type scheme \( \sigma \) is denoted as \( \sigma[\alpha \mapsto t] \). As shown by T-Inst-Row, a universally quantified row variable can only be instantiated by a row whose kind is the subset of the variable kind, equivalently speaking, by a "not more general" row. This rule ensures that well-formed rows in a type scheme are still well-formed after instantiation.

The typing rules for let-binding (T-Let) and the fixed point combinator (T-Fix) are standard. We use the strict version of the fixed point combinator in Typed ELEVATE.

The rule T-Label types a variant value with a label \( l \) and an expression \( e \). It can be considered as an inlined instantiation of the type scheme \( \forall (r : \neg\{l\}) \). \( \langle l : t \mid r \rangle \), hence the row kind \( \mathcal{R} \) is the subset of \( \neg\{l\} \).
\[ \Delta \vdash t : T \quad \alpha \not\in ftv(t) \]
\[ \Delta \vdash \forall (\alpha : T). \sigma \leq \sigma[\alpha \mapsto t] \]  
\begin{align*}
\text{T-Inst-Type} \\

\Delta \vdash \rho : R_{inst} \quad R_{inst} \subseteq R \quad \alpha \not\in ftv(\rho) \\
\Delta \vdash \forall (\alpha : R). \sigma \leq \sigma[\alpha \mapsto \rho] \\
\text{T-Inst-Row} \\

\Delta \vdash \sigma_0 \leq \sigma_1 \quad \Delta \vdash \sigma_1 \leq \sigma_2 \\
\Delta \vdash \sigma_0 \leq \sigma_2 \\
\text{T-Inst-Trans}
\end{align*}

Figure 3.7: Type scheme instantiation rules

\[ \Delta ; \Gamma \vdash e : \alpha \ as \ \tau \]
\[ \Delta ; \Gamma \vdash e : \tau[\alpha \mapsto \alpha \ as \ \tau] \]  
\begin{align*}
\text{T-Unfold} \\

\Delta ; \Gamma \vdash e : \tau \quad \Delta ; \Gamma \vdash e : \alpha \ as \ \tau \\
\text{T-Fold}
\end{align*}

Figure 3.8: Typing rules for equi-recursive types

Figure 3.8 gives the standard folding and unfolding rules for equi-recursive types.

Figure 3.9 is a collection of the typing rules for record operations. Unlike the type of the newly created variant value in rule T-Label, it is impossible to make the underlying row in the type of a newly created record contain more labels than the presented ones. Thus, T-RecordCons states that the type of the record ends with an empty row, which is the least general row. The rest of the rules are straightforward: field(s) should exist in the record to be modified, accessed, or deleted, and to extend a record, the to-be-added field(s) should not previously exist in the record.

Pattern matching is an essential part of Typed ELEVATE, and the corresponding typing rules are given in Figure 3.10. T-Void and T-Unit are defined in the standard way. The premises of T-Match generalize the type of the matched expression e, split the generalized type into two parts, and assign them to \( x_1 \) and \( x_2 \), respectively. Generalization during pattern matching is not common, but this behaviour can be found in OCaml and Castagna et al. [2016], and it allows more programs to typecheck. In other words, in Typed ELEVATE, pattern matching can be used as an advanced form of let-binding, which simultaneously introduces polymorphic variables and analyses different cases of an expression.
\[ \forall i \in \mathcal{N}, \Delta; \Gamma \vdash e_i : t_i \]  
\[ \Delta; \Gamma \vdash \{ (l_i : e_i)_{i \in \mathcal{N}} \} : \{ (l_i : t_i)_{i \in \mathcal{N}} \mid \cdot \} \]  
\[ \text{T-RecordCons} \]

\[ \Delta; \Gamma \vdash e : \{ (l_i : t_i)_{i \in \mathcal{N}} \mid \rho \} \]  
\[ \forall i \in \mathcal{N}, \Delta; \Gamma \vdash e_i : t_i \]  
\[ \Delta; \Gamma \vdash e . \{ (l_i : e_i)_{i \in \mathcal{N}} \} : \{ (l_i : t_i)_{i \in \mathcal{N}} \mid \rho \} \]  
\[ \text{T-RecordMod} \]

\[ \Delta; \Gamma \vdash e : \{ \rho \} \quad \Delta \vdash \rho : \mathcal{R} \quad \mathcal{R} \subseteq -\{ (l_i)_{i \in \mathcal{N}} \} \]  
\[ \forall i \in \mathcal{N}, \Delta; \Gamma \vdash e_i : t_i \]  
\[ \Delta; \Gamma \vdash e . + . \{ (l_i : e_i)_{i \in \mathcal{N}} \} : \{ (l_i : t_i)_{i \in \mathcal{N}} \mid \rho \} \]  
\[ \text{T-RecordExt} \]

\[ \Delta; \Gamma \vdash e : \{ l : t \mid \rho \} \]  
\[ \Delta; \Gamma \vdash e.l : t \]  
\[ \text{T-FieldAccess} \]

\[ \Delta; \Gamma \vdash e : \{ l : t \mid \rho \} \]  
\[ \Delta; \Gamma \vdash e : \{ \rho \} \]  
\[ \Delta; \Gamma \vdash e : \{ \cdot \Rightarrow \cdot \} \Rightarrow \{ \cdot \} \Rightarrow t_{\text{rhs}} \]  
\[ \text{T-FieldDel} \]

\[ \Delta; \Gamma \vdash e : \langle \cdot \rangle \]  
\[ \Delta; \Gamma \vdash \text{match } e \text{ with } \langle \cdot \rangle : t_{\text{rhs}} \]  
\[ \text{T-Void} \]

\[ \Delta; \Gamma \vdash e : \{ \cdot \} \quad \Delta; \Gamma \vdash \text{match } e \text{ with } \langle \cdot \rangle \Rightarrow t_{\text{rhs}} : t_{\text{rhs}} \]  
\[ \text{T-Unit} \]

\[ (\alpha_i : \kappa_i)_{i \in \mathcal{N}} = \text{ftv}(t) \setminus \text{ftv}(\Gamma) \]  
\[ (\alpha_j : \kappa_j)_{j \in \mathcal{M}} = \text{ftv}(\langle \rho \rangle) \setminus \text{ftv}(\Gamma) \]  
\[ \Delta, (\alpha_i : \kappa_i)_{i \in \mathcal{N}} \cup (\alpha_j : \kappa_j)_{j \in \mathcal{M}}; \Gamma \vdash e : \{ l : t \mid \rho \} \]  
\[ \sigma_{x_1} = (\forall (\alpha_i : \kappa_i).)_{i \in \mathcal{N}} t \quad \sigma_{x_2} = (\forall (\alpha_j : \kappa_j).)_{j \in \mathcal{M}} \langle \rho \rangle \]  
\[ \Delta; \Gamma, x_1 : \sigma_{x_1} \vdash \text{rhs}_1 : t_{\text{rhs}} \quad \Delta; \Gamma, x_2 : \sigma_{x_2} \vdash \text{rhs}_2 : t_{\text{rhs}} \]  
\[ \Delta; \Gamma \vdash \text{match } e \text{ with } \langle l \ x_1 \Rightarrow \text{rhs}_1 \mid x_2 \Rightarrow \text{rhs}_2 \rangle : t_{\text{rhs}} \]  
\[ \text{T-Match} \]
3.3.3 Operational Semantics

The operational semantics of Typed ELEVATE is given by Figure 3.11 and Figure 3.12 following the style used in Wright and Felleisen [1994]. Figure 3.11 provides the definitions of Typed ELEVATE values and the evaluation contexts. Figure 3.12 provides the reduction rules (named with the prefix ST-) and the stepping relation (denoted by \( \rightarrow \)) for the small-step operational semantics of Typed ELEVATE. All the rules are straightforward.

3.3.4 Properties of the Type System

With the static and dynamic semantics defined, we now show the soundness of the type system using the usual subject reduction and progress lemmas.

**Lemma 3.3.1** (Subject Reduction). If \( \Delta; \cdot \vdash e_1 : t \), and \( e_1 \rightsquigarrow e_2 \), then \( \Delta; \cdot \vdash e_2 : t \).

**Lemma 3.3.2** (Progress). If \( \Delta; \cdot \vdash e : t \), then either \( e \) is a value, or there exists an \( \hat{e} \) such that \( e \rightsquigarrow \hat{e} \).

**Theorem 3.3.3** (Type Soundness). If \( \Delta; \cdot \vdash e : t \), then either \( e \) is a value, or there exists an \( \hat{e} \) such that \( e \rightsquigarrow \hat{e} \) and \( \Delta; \cdot \vdash \hat{e} : t \).

The proofs can be found in Section 3.A.

To be more specific, this type system guarantees that strategies that type check:

- do not fail at runtime due to a missing case in pattern matching (guaranteed by the type soundness theorem); and
\[(\lambda x = e) \mapsto e[x \mapsto v] \quad \text{ST-App}\]

\[\begin{align*}
\text{let } f = v \text{ in } e_2 \mapsto e_2[f \mapsto v] & \quad \text{ST-Let} \\
\text{fix } v \mapsto v & \quad \text{ST-Fix} \\
\{ (l_i : v_i)_{i \in M} \mid l : v \mid (l_j : v_j)_{j \in N} \}. l \mapsto v & \quad \text{ST-FieldAccess} \\
\{ (l_i : v_i)_{i \in M} \mid (l_j : v_j)_{j \in N} \}. + \{ (l_i : v_i')_{i \in M} \mid (l_j : v_j)_{j \in N} \} & \quad \text{ST-RecordExt} \\
\end{align*}\]

\[\begin{align*}
\text{match } \text{} \text{with } \langle \cdot \Rightarrow e \rangle \mapsto e & \quad \text{ST-Match-Unit} \\
\text{match } l \; v \text{ with } \langle l \; x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle \mapsto e_1[x_1 \mapsto v] & \quad \text{ST-Match-Match} \\
\text{match } l \; v \text{ with } \langle l' \; x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle \mapsto e_2[x_2 \mapsto l \; v] & \quad \text{ST-Match-Skip} \\
E[e_1] \mapsto E[e_2] \text{ iff } e_1 \mapsto e_2 & \quad \text{S-Context} \\
\end{align*}\]

Figure 3.12: Small-step operational semantics of Typed ELEVATE

- do not access a non-existent field in a record (guaranteed by the type soundness theorem); and

- do not contain records or variants involving fields or cases tagged by the same label (guaranteed by the kinding and well-formedness rules); and

- do not contain a **dead branch** in pattern matching that is statically guaranteed not to be reached.

**On the detection of dead branches**

By only relying on the type system we are able to detect two forms of dead branches, which are unreachable cases in pattern matching expressions:

- A dead branch is detected when a label used in the pattern has already been matched by previous branches or cannot occur in the type of the matched expression. For example:

  ★ when the same label \(A\) is repeated multiple times:

  \[\text{match } x \text{ with } \langle A \Rightarrow \text{rhs} \mid y \Rightarrow \text{match } y \text{ with } \langle A \Rightarrow \text{dead_code} \mid \ldots \rangle \rangle\]
★ when \( x \) with type \( \langle B : \{ \cdot \} \mid r \rangle \) and where the kind of \( r \) is \( \neg \{ A, B \} \) is matched against label \( A \):

\[
\text{match } x \text{ with } \langle A \Rightarrow \text{dead code} \mid \ldots \rangle
\]

- A dead branch is detected when a row variable is exploited that is only there for type compatibility purposes as discussed earlier in Section 3.2. Formally, these kind of row variables do not occur free in the typing context. They do not contain meaningful information (for pattern matching) and can always be substituted by the empty row. Examples ruled out by this include:

★ the inferred type for \( \text{True} \) is \( \langle \text{True} : \{ \cdot \} \mid r \rangle \) where the kind of \( r \) is \( \neg \{ \text{True} \} \) but \( r \) does not occur free in the typing context. This disallows matching the value \( \text{True} \) against the pattern \( \text{False} \):

\[
\text{match } \text{True with } \langle \text{False } \Rightarrow \text{dead code} \mid \ldots \rangle
\]

★ similarly, we disallow matching the remainder \( x \) after the \( \text{True} \) branch against any other types except the empty variant. The variable \( x \) has type \( \langle r \rangle \) but \( r \) does not occur free in the typing context and is substituted with the empty row, ruling out expressions in the following form where the expected type of \( x \) is not \( \langle \cdot \rangle \):

\[
\text{match } \text{True with } \langle \text{True } \Rightarrow \text{rhs}_0 \mid x \Rightarrow \text{dead code} \rangle
\]

There are other forms of dead branches that our current type system is unable to detect. For example, the inferred type of \( (\text{seq fail id}) \) is \( t_0 \rightarrow \langle \text{Failure} : \{ \cdot \} \mid \text{Success} : t_1 \mid r \rangle \), but we know that the result can never be \( \text{Success} \). Nevertheless, we have to deal with the \( \text{Success} \) case when we analyze the result with pattern matching.

However, the type actually tell us that the \( \text{Success} \) case is unnecessary because \( t_1 \) does not occur free in the typing context and it can be substituted by the empty type – since there is only one way to use the empty type, a branch for the case is useless. This particular situation, is similar to the second form of dead code detection mentioned above. The underlying common idea is that, if a type \( t \) contains type variables which do not occur free in the typing context, and substituting these type variables with empty type/row will make \( t \) isomorphic to the empty type, then there is no need to deal with a value of this type in pattern matching. In the general case, allowing the types which are isomorphic to the empty type to be used as empty type require a more complex type system and put forward challenges for type inference. A possible solution to this is to use semantic subtyping as in Castagna et al. [2016], which gives
a complete and sound algorithm to solve type constraints with semantic subtyping, in the price of losing principal solutions.

3.4 Implementation

After introducing the type system formally in Section 3.3 we now discuss its practical implementation focusing on two important aspects: pattern elaboration and type inference.

3.4.1 Pattern Elaboration

As mentioned in Sections 3.2.2 and 3.3.2, pattern matching is an important part of Typed ELEVATE, but patterns that are easy to write for programmers may be too complex for type inference and exhaustiveness checking. Many languages [Morris and McKinna, 2019; Hillerström and Lindley, 2016; Gaster, 1998; Pottier and Rémy, 2004c] bypass or do not consider performing type inference and exhaustiveness checking for deeply nested patterns by only including shallow / simple patterns in the formalization, and bridge the gap (if exists) with syntactical transformation [Gaster, 1998]. This is also the case for Typed ELEVATE, and pattern elaboration is used to bridge this gap between the formalized Typed ELEVATE and its practical implementation.

| Simple Patterns | \( \pi := x | l | \{ \cdot \} \) |
| Complex Patterns | \( \tilde{\pi} := x | l[\tilde{\pi}] | \{(l_i : \tilde{\pi}_i)_{i \in \mathbb{N}} \} \) |
| Field Access Forms | \( \delta := x[l][\cdot] \) |
| Match IDs | \( \theta := \mathbb{N} | \theta | \mathbb{N} \) |
| Match Chains | \( \omega := e^\theta \mathbf{match}^\theta \delta \mathbf{with} \{(\pi_i \Rightarrow \omega_i)_{i \in \mathbb{N}}\} \) |

Figure 3.13: Syntax of simple patterns, complex patterns, field access forms, match IDs and match chains

Figure 3.13 shows the abstract syntax of simple patterns (denoted by \( \pi \)) and complex patterns (denoted by \( \tilde{\pi} \)). The syntax of simple patterns are extracted from the \textbf{match} expressions in Figure 3.1 and can only be a variable pattern (denoted by \( x \)), a label \( l \) followed by a variable (denoted by \( l x \)), or a unit pattern (denoted by \( \{ \cdot \} \)). In comparison, complex patterns allow the recursive occurrence of complex patterns inside patterns.
Chapter 3. A Row-Polymorphic Type System for Strategic Rewriting

Listing 3.5: The pseudo-code of pattern elaboration

```plaintext
1 patElab(match x with ⟨(π_i ⇒ rhs_i)⟩) =
2   desugar(refine(x → x; foldl1(merge,
3   (sort(patExpn(θ; 0; u; match x with ⟨π_i ⇒ rhs_i⟩))}_{i ∈ N}))
```

a label (denoted by \( l \tilde{π} \)) and the usage of record patterns (denoted by \( \{(l_i : \tilde{π}_i)\}_{i ∈ N} \)), which match the distinct fields \( (l_i)_{i ∈ N} \) respectively with complex patterns \( (\tilde{π}_i)_{i ∈ N} \). As in the formalization, the order of the fields is insignificant in a record pattern. Linearity checks will be performed to make sure that each variable only appears once in a complex pattern.

For a single pattern matching expression with more than one branch, the basic idea of pattern elaboration is to convert the complex pattern in each branch into nested pattern matching expressions only using (simple) patterns, and then merging all branches to generate a decision tree [Maranget, 2008].

Listing 3.5 shows the pseudo-code of pattern elaboration. Functions `patExpn` and `merge` perform the conversion and merging mentioned above, respectively. Function `foldl1` performs left-folding of list and takes the first element of the list as the starting value. Function `sort` rearranges the order of the nested pattern matching expressions generated by `patExpn` to get a more efficient result. Since the efficiency of pattern matching is not of major concern in this work, the `sort` function will not be discussed in details here. Finally, function `refine` adjusts expressions in the decision tree to get a more precise type inference result and `desugar` convert match chains into ordinary pattern matching expressions.

Listing 3.6 shows the pseudo-code of `patExpn` which means "pattern expansion". Specifically, a complex pattern will expand into a series of simple pattern matching expressions. A key part of this process is the conversion of record patterns. Since a record pattern is just multiple patterns put together, where each one of them is associated with a label, a straightforward method is replacing a record pattern with a variable pattern \( x \), then selecting each matched field of \( x \), and then recursively performing the conversion for each field. Similar conversion can also be applied to complex label patterns \( (l \tilde{π}) \). This method naturally covers a special case of the record patterns, namely \( {} \): it does not select any field to match, so it matches all records and is directly replaced by a variable pattern during the conversion. An awkward consequence is that a variable pattern does not only match records, but also matches other
Listing 3.6: The pseudo-code of pattern expansion

type of values, so there must be some other way to distinguish an ordinary variable pattern and \{\}. This leads us to the design of the field access forms whose syntax is shown in Figure 3.13. A field access form (denoted by $\delta$) can be a variable ($x$) or a field access ($x.l$), optionally followed by an empty record modification ($x.\{\}$ or $x.l.\{\}$). As its name suggests, the field access form expresses the intermediate field accesses required to convert record patterns. Since the empty record modification does nothing but enforce the modified value to be a record, it helps distinguish ordinary variable patterns and variable patterns which used to be record patterns.

With the conversion above, each single branch of a pattern matching expression can be converted to a biased decision tree, and we use match chains to encode this. Figure 3.13 shows the syntax of match chains. A match chain (denoted by $\omega$) is a multi-way tree-like structure used throughout the pattern elaboration process. It can be either an ordinary expression, i.e., the RHS (Right-Hand Side) expression, or a pattern matching expression whose RHS expressions are match chains. Since there can be multiple complex patterns in one record pattern, and a complex pattern can be deeply nested or even repeatedly occurs inside itself, to identify the depths and locations of match chain nodes, a match ID is assigned to each of them. Figure 3.13
includes the syntax of match IDs. A match ID (denoted by \( \theta \)) is simply a non-empty sequence of natural numbers whose length indicates the depth in a nested complex pattern and the exact numbers indicate the locations in record patterns.

```plaintext
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \mid \ell \Rightarrow \omega_a \rangle; \text{match}^b \delta_b \text{ with } \langle x_b \Rightarrow \omega_b \rangle) \\
\text{when length}(\ell_b) = \text{length}(\ell_b) \text{ and } \delta_a \simeq \delta_b = \text{match}^a \delta_b \text{ with } (\pi_i \Rightarrow \text{merge}(\omega_i; \omega_b[x_b \mapsto \pi_i]))_{i \in N} \mid x_a \Rightarrow \text{merge}(\omega_a; \omega_b[x_b \mapsto x_a])) \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \rangle; \text{match}^b \delta_b \text{ with } \langle x_b \Rightarrow \omega_b \rangle) \\
\text{when length}(\ell_a) = \text{length}(\ell_b) \text{ and } \delta_a \simeq \delta_b = \text{match}^a \delta_b \text{ with } (\pi_i \Rightarrow \text{merge}(\omega_i; \omega_b[x_b \mapsto \pi_i]))_{i \in N} \mid x_b \Rightarrow \omega_b) \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \mid l \Rightarrow \omega_a \mid (\pi_j \Rightarrow \omega_j)_{j \in M} \rangle; \text{match}^b \delta_b \text{ with } \langle l \Rightarrow \omega_b \rangle) \\
\text{when length}(\ell_a) = \text{length}(\ell_b) \text{ and } \delta_a \simeq \delta_b = \text{match}^a \delta_b \text{ with } (\pi_i \Rightarrow \omega_i)_{i \in N} \mid l \Rightarrow \text{merge}(\omega_a; \omega_b) \mid (\pi_j \Rightarrow \omega_j)_{j \in M}) \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \rangle; \text{match}^b \delta_b \text{ with } \langle \pi_b \Rightarrow \omega_b \rangle) \\
\text{when length}(\ell_a) = \text{length}(\ell_b) \text{ and } \delta_a \simeq \delta_b = \text{match}^a \delta_b \text{ with } (\pi_i \Rightarrow \omega_i)_{i \in N} \mid x_a \Rightarrow \text{merge}(\omega_a; \text{match}^b \ell_a \text{ with } \langle \pi_b \Rightarrow \omega_b \rangle) \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \rangle; \text{match}^b \delta_b \text{ with } \langle \pi_b \Rightarrow \omega_b \rangle) \\
\text{when length}(\ell_a) = \text{length}(\ell_b) \text{ and } \delta_a \simeq \delta_b = \text{match}^a \delta_b \text{ with } (\pi_i \Rightarrow \omega_i)_{i \in N} \mid x_b \Rightarrow \omega_b) \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \rangle; \text{match}^b \delta_b \text{ with } \langle \pi_b \Rightarrow \omega_b \rangle) \\
\text{when length}(\ell_a) = \text{length}(\ell_b) = \text{match}^a \delta_b \text{ with } (\pi_i \Rightarrow \omega_i)_{i \in N} \mid x \Rightarrow \text{match}^b \delta_b \text{ with } \langle \pi_b \Rightarrow \omega_b \rangle) \text{ where } x \text{ is fresh} \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \rangle; \omega_b) = \\
merge(\text{match}^a \delta_a \text{ with } \langle (\pi_i \Rightarrow \omega_i)_{i \in N} \rangle; \text{match}^a \delta_a \text{ with } (x \Rightarrow \omega_b)) \\
\text{where } x \text{ is fresh} \\
merge(e^{\theta_e}; \omega_b) = e^{\theta_e}
```

Listing 3.7: The pseudo-code of match chain merging

Listing 3.7 shows the pseudo-code of match chain merging. With match IDs and field access forms, \textit{merge} can easily tell if two match chains are matching the same value (hence can be merged). It should be noted that the second and some other lines of \textit{merge} duplicate \( \omega_b \), while the final line of \textit{merge} removes \( \omega_b \). This is related with another usage of the match IDs: they are also the unique identifiers for RHS.
refine(S; match^θ δ with \((\pi_i \Rightarrow \omega_i)_{i \in N}\)) =

match^θ δ with \((\text{refineStep}(\delta; \pi_i \Rightarrow \omega_i))_{i \in N}\)

where refineStep(x; \pi ⇒ \omega) = \pi ⇒ \text{refine}((x \mapsto \pi) \circ S; \omega_i)

refineStep(x.|{}; \pi ⇒ \omega) = \pi ⇒ \text{refine}((x \mapsto \omega + \{l : \pi\}) \circ S; \omega_i)

refineStep(x.l.|{}; \pi ⇒ \omega) = \pi ⇒ \text{refine}((x \mapsto x.l. + \{l : \pi\}) \circ S; \omega_i)

refine(S; e^θ) = e^θ[S]

Listing 3.8: The pseudo-code of match chain refining

desugar(match^θ δ with \((l \Rightarrow \omega) \mid (\pi_i ⇒ \omega_i)_{i \in N}\)) =

desugar \((\text{match \; } \delta \; \text{ with } \langle \{\} \Rightarrow \text{desugar}() \rangle \mid \text{r \Rightarrow \text{desugar(match^θ \; \text{r \; with } \langle (\pi_i ⇒ \omega_i)_{i \in N}\rangle))})\)

where x and r are fresh

desugar(match^θ δ with \((l \; x \Rightarrow \text{match \; x \; with } \langle \{\} \Rightarrow \text{desugar}() \rangle \mid \text{r \Rightarrow \text{desugar(match^θ \; \text{r \; with } \langle (\pi_i ⇒ \omega_i)_{i \in N}\rangle))})\)

where r is fresh

desugar(match^θ δ with \((x ⇒ \omega)\)) = \text{let \; x = δ \; in \; desugar(\omega)}

desugar(match^θ δ with \())\) = match δ with \()

desugar(e^θ) = e^θ[

Listing 3.9: The pseudo-code of match chain desugaring

expressions. All the duplications and removals of RHS expressions are traced, and if all the occurrences of a RHS expression are removed, a redundant patterns error will be reported.

Listing 3.8 shows the pseudo-code of match chain refining. The idea behind this definition is straightforward. If a field access form \(\delta\) is matched by a pattern \(\pi\), we know that in the corresponding RHS expression, the actual value of \(\delta\) can only be the expression counterpart of \(\pi\). To get a more precise type inference result, \text{refine} substitutes the identifier in \(\delta\) with \(\pi\) or the corresponding record operations.

Finally, Listing 3.9 shows the pseudo-code of match chain desugaring, which recursively converts each branch in a match chain into cascaded pattern matching expressions.
3.4.2 Type Inference

The type inference of Typed ELEVATE follows the widely used Hindley-Milner style [Milner, 1978; Damas and Milner, 1982] and extends it with row polymorphism. The implementation of the core part of this inference algorithm, a union-find based unifier supporting equi-recursive types and rows, is largely based on the Huet’s unification algorithm [Knight, 1989; Huet, 1976] and the unifier implementation in the Mini inference engine by Pottier and Rémy [2004d]. Currently, Typed ELEVATE does not put any other restrictions besides contractiveness on the form of equi-recursive types.

Compared with the typing rules presented in Section 3.3.2, the type inference algorithm of Typed ELEVATE is more restrictive for pattern matching expressions. As discussed in Section 3.3.4, pattern matching branches may become dead code if some special row variables are substituted by empty rows. These dead branches will be removed during type inference, even if they can have valid type derivations using the rules for type checking. The RHS tracing mechanism mentioned in Section 3.4.1 still works here, and it will report errors when the occurrence count of a RHS expression is reduced to zero, otherwise the branch will be removed silently.

Using our practical implementation, we have implemented some examples from Hagedorn et al. [2020] in Typed ELEVATE observing the expected types by running type inference on them and confirming that the presented strategies are well typed.

3.5 Conclusion

In this chapter, we have presented a strategy language Typed ELEVATE with a row-polymorphic type system. We have presented its formal definition, its practical implementation, and a case study of ensuring the safe composition of program transformations. This type system guarantees that strategies that typecheck do not fail due to a missing case in pattern matching and do not contain dead branches.

However, this type system is not precise enough for detecting strategy compositions which always fail at runtime. In the next chapter, we will discuss the limitations of Typed ELEVATE in detail, and present a type system overcoming these limitations and preserving as many of the useful features of Typed ELEVATE as possible, where a novel concept called tracing is introduced.
Appendix 3.A  Type Soundness Proof

Lemma 3.A.1 (Type Extension). If $\Delta; \Gamma \vdash e : t$, and $\Gamma'(x) = \Gamma(x)$ for all $x \in f\nu(e)$, and $\Gamma'$ is well-formed with respect to $\Delta$, then $\Delta; \Gamma' \vdash e : t$.

Proof. By induction on the typing derivation of $\Delta; \Gamma \vdash e : t$. \hfill \Box

Lemma 3.A.2 (Kind Extension). If $\Delta; \Gamma \vdash e : t$, and $\Delta'(\alpha) = \Delta(\alpha)$ for all $\alpha \in f\tau(\Gamma) \cup f\tau(t)$, then $\Delta'; \Gamma \vdash e : t$.

Proof. By induction on the kinding derivation of $t$ and all types in $\Gamma$. \hfill \Box

Lemma 3.A.3 (Type Substitution). If $\Delta; \Gamma \vdash e : t$, and $\mathcal{S}$ is a substitution where $Reg(\mathcal{S}) \subseteq \Delta$, and $\Gamma[\mathcal{S}]$ and $t[\mathcal{S}]$ is well-formed with respect to $\Delta$, then $\Delta; \Gamma[\mathcal{S}] \vdash e : t[\mathcal{S}]$.

Proof. By induction on the typing derivation of $\Delta; \Gamma \vdash e : t$. \hfill \Box

Lemma 3.A.4 (Generalization). If $\Delta; \Gamma, x : \sigma \vdash e : t$, and $\Delta \vdash \sigma' \leq \sigma$, then $\Delta; \Gamma, x : \sigma' \vdash e : t$.

Proof. By induction on the typing derivation of $\Delta; \Gamma, x : \sigma \vdash e : t$. \hfill \Box

Lemma 3.A.5 (Substitution). If $\Delta; \Gamma, x : (\forall (a_i : \kappa_i))_{i \in N} t \vdash e : t'$, and $\Delta, (a_i : \kappa_i)_{i \in N} ; \vdash v : t$, and $(a_i)_{i \in N} \cap f\tau(\Gamma) = \emptyset$, then $\Delta; \Gamma \vdash e[x \mapsto v] : t'$.

Proof. By induction on the derivation of $\Delta; \Gamma, x : (\forall (a_i : \kappa_i))_{i \in N} t \vdash e : t'$

- Case $e = x'$
  
  ★ When $x' \neq x$, we have $\Delta; \Gamma \vdash x' : t'$ by $T$-Var, so $\Delta; \Gamma \vdash x'[x \mapsto v] : t'$.

  ★ When $x' = x$, we have $\Delta \vdash (\forall (a_i : \kappa_i))_{i \in N} t \leq t'$ and $(a_i)_{i \in N} \cap f\tau(t') = \emptyset$ by $T$-Var and the rules for $\leq$, which means

  \[
  \exists \mathcal{S}, Dom(\mathcal{S}) = (a_i)_{i \in N}, t[\mathcal{S}] = t'
  \]

  , then we have $\Delta, (a_i : \kappa_i)_{i \in N} ; \vdash v : t[\mathcal{S}]$ by Lemma 3.A.3, which simplifies to $\Delta, (a_i : \kappa_i)_{i \in N} ; \vdash v : t'$, hence $\Delta, (a_i : \kappa_i)_{i \in N} ; \Gamma \vdash x'[x \mapsto v] : t'$. Thus, we have $\Delta; \Gamma \vdash x'[x \mapsto v] : t'$ by assumptions and Lemma 3.A.2.
Chapter 3. A Row-Polymorphic Type System for Strategic Rewriting

**Case** \( e = (\lambda x' = e_1) \)

We have \( \Delta; \Gamma, x : (\forall (\alpha_i : \kappa_i)_{i \in N} t, x' : t_0 + e_1 : t_1 \) and \( t' = t_0 \rightarrow t_1 \) by T-Lam, and we construct a substitution \( S = (\alpha_i)_{i \in N} \leftrightarrow (\alpha'_i)_{i \in N} \) where \( (\alpha'_i)_{i \in N} \) are distinct from all existing type variables. Let \( \sigma = (\forall (\alpha_i : \kappa_i)_{i \in N} t, \) then we have

\[
\Delta, (\alpha'_i : \kappa'_i)_{i \in N}; \Gamma, x' : t_0, x : \sigma + e_1 : t_1 \quad (3.1)
\]

by Lemma 3.A.2 and Lemma 3.A.1,

\[
\Delta, (\alpha'_i : \kappa'_i)_{i \in N}; \Gamma, x' : t_0[S], x : \sigma + e_1 : t_1[S] \quad (3.2)
\]

by Lemma 3.A.3. On the other hand, we have

\[
\Delta, (\alpha'_i : \kappa'_i)_{i \in N}; (\alpha_i : \kappa_i)_{i \in N}; \cdot \vdash v : t \quad (3.3)
\]

by Lemma 3.A.2, and

\[
(\alpha_i)_{i \in N} \cap ftv(\Gamma, x' : t_0[S]) = \emptyset \quad (3.4)
\]

by the assumption about \( S \). Thus,

\[
\Delta, (\alpha'_i : \kappa'_i)_{i \in N}; \Gamma, x' : t_0[S] + e_1[x \mapsto v] : t_1[S]
\]

by the induction hypothesis with (3.2), (3.3) and (3.4). \( S \) is bijective, so it can be inverted and we get

\[
\Delta, (\alpha'_i : \kappa'_i)_{i \in N}; \Gamma, x' : t_0 + e_1[x \mapsto v] : t_1
\]

by Lemma 3.A.3, and then

\[
\Delta; \Gamma, x' : t_0 + e_1[x \mapsto v] : t_1
\]

by Lemma 3.A.2. Finally, we have

\[
\Delta; \Gamma + (\lambda x' \Rightarrow e_1)[x \mapsto v] : t'
\]

by T-Lam.

**Case** \( e = \textbf{let} \ f = e_1 \ \textbf{in} \ e_2 \)

Let \( \sigma = (\forall (\alpha_i : \kappa_i)_{i \in N} t, \) we have

\[
\Delta, (\alpha'_j : \kappa'_j)_{j \in M}; \Gamma, x : \sigma + e_1 : t_1 \quad (3.5)
\]
where \((\alpha\prime_j\big)_\in\mathcal{M} = ftv(t_1) \setminus ftv(\Gamma, x : \sigma)\) by T-Let. Let \(\Delta' = (\alpha\prime_j : \kappa\prime_j\big)_\in\mathcal{M}\), then we have

\[
\Delta, \Delta', (\alpha_i : \kappa_i)_\in\mathcal{N}; \vdash v : t
\]

(3.6)

by Lemma 3.A.2, and we get

\[
\Delta, \Delta'; \Gamma \vdash e_1[x \mapsto v] : t_1
\]

(3.7)

by the induction hypothesis with (3.5) and (3.6). On the other hand, let \(\sigma\prime_f = \left(\forall (\alpha\prime_j : \kappa\prime_j)\right)_\in\mathcal{T_1}\) t, we have

\[
\Delta; \Gamma, f : \sigma\prime_f, x : \sigma \vdash e_2[x \mapsto v] : t'
\]

(3.8)

by T-Let and Lemma 3.A.1, and

\[
(\alpha_i)_\in\mathcal{N} \cap ftv(\Gamma, f : \sigma\prime_f) = \emptyset
\]

(3.9)

by the assumption \((\alpha_i)_\in\mathcal{N} \cap ftv(\Gamma) = \emptyset\) and the algebra of sets. Thus, we get

\[
\Delta; \Gamma, f : \sigma\prime_f, x : \sigma \vdash e_2[x \mapsto v] : t'
\]

(3.10)

by the induction hypothesis with (3.8) and (3.9).

However, we cannot jump to the conclusion now because \(\sigma\prime_f\) is generalized with respect to \(\Gamma, x : \sigma\) instead of \(\Gamma\). Let \((\hat{\alpha}_p\big)_\in\mathcal{U} = ftv(t_1) \setminus ftv(\Gamma)\), and we know that

\[
(\alpha\prime_j\big)_\in\mathcal{M} \subseteq (\hat{\alpha}_p\big)_\in\mathcal{U},
\]

so there exists \(\hat{\alpha}\) and \((\kappa\''_p\big)_\in\mathcal{U}\) such that

\[
(\Delta, \Delta') = (\hat{\alpha}, (\hat{\kappa}_p\big)_\in\mathcal{U})
\]

Then we construct a substitution \(\mathcal{S} = (\hat{\alpha}_p\big)_\in\mathcal{U} \mapsto (\alpha\''_p\big)_\in\mathcal{U}\), where \((\alpha\''_p\big)_\in\mathcal{U}\) are distinct from all existing type variables. Let \(\Delta'' = (\alpha\''_p : \kappa\''_p\big)_\in\mathcal{U}\). On one hand, we have

\[
\Delta, \Delta', \Delta''; \Gamma \vdash e_1[x \mapsto v] : t_1[\mathcal{S}]
\]

by (3.7), Lemma 3.A.2 and Lemma 3.A.3, then we safely remove \(\Delta\) and get

\[
\Delta, \Delta''; \Gamma \vdash e_1[x \mapsto v] : t_1[\mathcal{S}]
\]

(3.11)

by Lemma 3.A.2. On the other hand, let

\[
\sigma\''_f = \left(\forall (\alpha\''_p : \kappa\''_p)\right)_\in\mathcal{U} t_1[\mathcal{S}]
\]
, we have

$$\Delta;\Gamma, f : \sigma''_f \vdash e_2[x \mapsto v] : t'$$

(3.12)

by (3.10) and Lemma 3.A.4. Finally, we get

$$\Delta;\Gamma \vdash (\text{let } f = e_1 \text{ in } e_2)[x \mapsto v] : t'$$

by (3.11), (3.12) and T-Let.

• Case \(e = \text{match } e_1 \text{ with } \langle \text{let } x_1 \mapsto \text{rhs}_1 \mid x_2 \mapsto \text{rhs}_2 \rangle\)

Let \(\sigma = (\forall (\alpha_i : \kappa_i), i \in N) \cdot t\), we have

$$\Delta_i \left(\alpha'_j : \kappa'_j\right)_{j \in M} : \Gamma, x : \sigma \vdash e_1 : \langle l : t_1 \mid \rho \rangle$$

(3.13)

where \(\left(\alpha'_j\right)_{j \in M} = \left(\alpha'_p\right)_{p \in U} \cup \left(\alpha'_q\right)_{q \in V} = (ftv(t_1) \setminus ftv(\Gamma, x : \sigma)) \cup (ftv(\langle \rho \rangle) \setminus ftv(\Gamma, x : \sigma))\) by T-Match. Let \(\Delta' = \left(\alpha'_j : \kappa'_j\right)_{j \in M}\), then we have

$$\Delta, \Delta', (\alpha_i : \kappa_i)_{i \in N} \vdash \nu : t$$

(3.14)

by Lemma 3.A.2, and we get

$$\Delta, \Delta'; \Gamma \vdash e_1[x \mapsto v] : \langle l : t_1 \mid \rho \rangle$$

(3.15)

by the induction hypothesis with (3.13) and (3.14). On the other hand, let \(\sigma'_{x_1} = \left(\forall (\alpha'_p : \kappa'_p)\right)_{p \in U} t_1\) and \(\sigma'_{x_2} = \left(\forall (\alpha'_q : \kappa'_q)\right)_{q \in V} \langle \rho \rangle\), we have

$$\Delta;\Gamma, x_1 : \sigma'_{x_1}, x : \sigma \vdash \text{rhs}_1 : t'$$

(3.16)

and

$$\Delta;\Gamma, x_2 : \sigma'_{x_2}, x : \sigma \vdash \text{rhs}_2 : t'$$

(3.17)

by T-Let and Lemma 3.A.1. Thus, we get

$$\Delta;\Gamma, x_1 : \sigma'_{x_1} \vdash \text{rhs}_1[x \mapsto v] : t'$$

(3.18)

and

$$\Delta;\Gamma, x_2 : \sigma'_{x_2} \vdash \text{rhs}_2[x \mapsto v] : t'$$

(3.19)

by the induction hypothesis with (3.16) and (3.17).

Still, more proof steps are required because \(\sigma'_{x_1}\) and \(\sigma'_{x_2}\) are not generalized with respect to \(\Gamma, x : \sigma\), but the same proof technique for \(e = \text{let } f = e_1 \text{ in } e_2\) can be used here, so part of the proof is omitted. Let \(S\) and \(\Lambda''\) be the newly constructed
substitution and kinding environment, and $\sigma'_{x_1}$ and $\sigma'_{x_2}$ be the more generalized type schemes, we have

$$\Delta, \Delta''; \Gamma \vdash e_1[x \mapsto v] : \langle l : t_1 \mid \rho \rangle \{S\}$$  \hspace{1cm} (3.20)

and

$$\Delta; \Gamma, x_1 : \sigma''_{x_1} \vdash \text{rhs}_1[x \mapsto v] : t'$$  \hspace{1cm} (3.21)

and

$$\Delta; \Gamma, x_2 : \sigma''_{x_2} \vdash \text{rhs}_2[x \mapsto v] : t'$$  \hspace{1cm} (3.22)

Finally, we get

$$\Delta; \Gamma \vdash (\text{match } e \text{ with } \langle l \ x_1 \Rightarrow \text{rhs}_1 \mid x_2 \Rightarrow \text{rhs}_2 \rangle)[x \mapsto v] : t'$$

by T-Match with (3.20), (3.21) and (3.22)

- For the rest of the cases, they can be routinely proven by applying the induction hypothesis. The proof for $e = \text{fun arg}$ is given here as an example: it follows from T-App and the induction hypothesis that

$$\Delta; \Gamma \vdash \text{fun}[x \mapsto v] : t_1 \rightarrow t'$$

$$\Delta; \Gamma \vdash \text{arg}[x \mapsto v] : t_1$$

, hence $\Delta; \Gamma \vdash \text{fun arg}[x \mapsto v] : t'$ by T-App.

\[ \square \]

Lemma 3.A.6 (Subject Reduction). If $\Delta; \cdot \vdash e_1 : t$, and $e_1 \rightsquigarrow e_2$, then $\Delta; \cdot \vdash e_2 : t$.

Proof. By case analysis on the reduction $e_1 \rightsquigarrow e_2$.

- **Case** $(\lambda x = e) \overset{v}{\mapsto} e[x \mapsto v]$

  We have $\Delta; \cdot \vdash (\lambda x = e) : t_1 \rightarrow t$ and $\Delta; \cdot \vdash v : t_1$ by T-App, and then we get $\Delta; x : t_1 \vdash e : t$ by T-Lam, and finally $\Delta; \cdot \vdash e[x \mapsto v] : t$ by Lemma 3.A.5.

- **Case** $\text{let } f = v \text{ in } e_2 \overset{f \mapsto v}{\mapsto} e_2[f \mapsto v]$

  We have $\Delta_1 (\alpha_i : \kappa_i)_{i \in N}; \cdot \vdash v : t_1$ where $(\alpha_i)_{i \in N} = \text{fst}(t_1)$, and

  $\Delta; f : (\forall (\alpha_i : \kappa_i).)_{i \in N} t_1 \vdash e_2 : t$ by T-Let, and then $\Delta; \cdot \vdash e_2[f \mapsto v] : t$ by Lemma 3.A.5.
• Case

\[
\textbf{match } l \cdot v \textbf{ with } \langle l \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle
\]
\[
\rightsquigarrow e_1 [x_1 \mapsto v]
\]

We have \(\Delta, (\alpha_i : \kappa_i)_{i \in N} ; \vdash l \cdot v : \langle l : t_1 \mid \rho \rangle\) where \((\alpha_i)_{i \in N} = (\alpha_p)_{p \in \mathcal{U}} \cup (\alpha_q)_{q \in \mathcal{V}} = f t v (t_1) \cup f t v (\langle \rho \rangle)\), and \(\Delta ; x_1 : (\forall (\alpha_p : \kappa_p))_{p \in \mathcal{U}} t_1 \vdash e_1 : t\) by T-Match, and then we get \(\Delta, (\alpha_p : \kappa_p)_{p \in \mathcal{U}} ; \vdash v : t_1\) by T-Label and Lemma 3.A.2, and finally \(\Delta ; \vdash e_1 [x_1 \mapsto v] : t\) by Lemma 3.A.5.

• Case

\[
\textbf{match } l \cdot v \textbf{ with } \langle l' \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \rangle
\]
\[
\rightsquigarrow e_2 [x_2 \mapsto l \cdot v]
\]

We have \(\Delta, (\alpha_i : \kappa_i)_{i \in N} ; \vdash l \cdot v : \langle l' : t_1 \mid \rho \rangle\) where \((\alpha_i)_{i \in N} = (\alpha_p)_{p \in \mathcal{U}} \cup (\alpha_q)_{q \in \mathcal{V}} = f t v (t_1) \cup f t v (\langle \rho \rangle)\), and \(\Delta ; x_2 : (\forall (\alpha_q : \kappa_q))_{q \in \mathcal{V}} \rangle t_1 \vdash e_2 : t\) by T-Match, and then we get \(\Delta, (\alpha_q : \kappa_q)_{q \in \mathcal{V}} ; \vdash v : \langle \rho \rangle\) where \(\rho = (l : t' \mid \rho')\) and \(\Delta, (\alpha_q : \kappa_q)_{q \in \mathcal{V}} ; \vdash v : t'\) by T-Label and Lemma 3.A.2, and finally \(\Delta ; \vdash e_2 [x_2 \mapsto l \cdot v] : t\) by Lemma 3.A.5.

• For the rest of the cases, they can be routinely proven by applying the typing rules and their inversions. The proof for \(\text{fix } v \rightsquigarrow v \ (\lambda x = \text{fix } v \ x)\) is given here as an example: we have \(t = t_1 \rightarrow t_2\) and \(\Delta ; \vdash v : (t_1 \rightarrow t_2) \rightarrow t_1 \rightarrow t_2\) by T-App and T-Fix, and then we have \(\Delta ; \vdash (\lambda x = \text{fix } v \ x) : t_1 \rightarrow t_2\) by T-App and T-Lam, and finally \(\Delta ; \vdash (\lambda x = \text{fix } v \ x) : t_1 \rightarrow t_2\) by T-App.

\(\square\)

**Lemma 3.A.7** (Canonical Forms). If \(v\) is a value,

- \(v = l \cdot v'\) if \(\Delta ; \vdash v : \langle l : t' \mid \rho \rangle\).
- \(v = \{ (l_i : v_i)_{i \in N} \}\) if \(\Delta ; \vdash v : \{ (l_i : t_i)_{i \in N} \}\).
- \(v = (\lambda x = e)\) or \(v = \text{fix}\) if \(\Delta ; \vdash v : t_1 \rightarrow t_2\).

**Proof.** By inversion on the typing judgement in each case. \(\square\)

**Lemma 3.A.8** (Progress). If \(\Delta ; \vdash e : t\), then either \(e\) is a value, or there exists an \(\hat{e}\) such that \(e \rightsquigarrow \hat{e}\).

**Proof.** This lemma can be restated as: if \(\Delta ; \vdash e : t\), then either \(e\) is a value, or there exist an evaluation context \(E\) and two expressions \(e'\) and \(e''\) such that \(e = E[e']\) and \(e' \rightsquigarrow e''\).

By induction on the derivation of \(\Delta ; \vdash e : t\).
• Case $e = x$
  Impossible.

• Case $e = (\lambda x = e_1)$
  $e$ is already a value.

• Case $e = \text{fun arg}$
  We have
  \[
  \Delta; \vdash \text{fun} : t_1 \rightarrow t \tag{3.23}
  \]
  and
  \[
  \Delta; \vdash \text{arg} : t_1 \tag{3.24}
  \]
  by T-App. By applying the induction hypothesis on (3.23), we get:
  
  ★ If $\text{fun} = E_{\text{fun}}[e']$, and $e' \rightsquigarrow e''$, then there exists $E = E_{\text{fun}} \text{ arg}$ such that $e = E[e']$.
  
  ★ If $\text{fun}$ is a value, by applying the induction hypothesis on (3.24), we get:
    
    * If $\text{arg} = E_{\text{arg}}[e']$, and $e' \rightsquigarrow e''$, then there exists $E = \text{fun} E_{\text{arg}}$ such that $e = E[e']$.
    
    * If $\text{arg}$ is a value, and $\text{fun}$ can be either $(\lambda x = e_1)$ or $\text{fix}$ by Lemma 3.A.7, then there exists a reduction with $E = []$ by ST-App and ST-Fix, respectively.

• Case $e = \text{let } f = e_1 \text{ in } e_2$
  We have
  \[
  \Delta, (\alpha_i : \kappa_i)_{i \in \mathbb{N}}; \vdash e_1 : t_1 \tag{3.25}
  \]
  where $(\alpha_i)_{i \in \mathbb{N}} = \text{fv}(t_1)$ by T-Let. By applying the induction hypothesis on (3.25), we get:
  
  ★ If $e_1 = E_1[e']$, and $e' \rightsquigarrow e''$, then there exists $E = \text{let } f = E_1 \text{ in } e_2$ such that $e = E[e']$.
  
  ★ If $e_1$ is a value, there exists a reduction with $E = []$ by ST-Let.

• Case $e = \text{fix}$
  $e$ is already a value.
• **Case** \( e = l \ e_1 \)

We have

\[
\Delta; \cdot \vdash e_1 : t_1
\]  
(3.26)

by T-Label. By applying the induction hypothesis on (3.26), we get:

★ If \( e_1 = E_1[e'] \), and \( e' \leadsto e'' \), then there exists \( E = l \ E_1 \) such that \( e = E[e'] \).

★ If \( e_1 \) is a value, then \( e \) is a value.

• **Case** \( e = \text{match} \ e \ \text{with} \ \langle \rangle \)

We have

\[
\Delta; \cdot \vdash e_1 : \langle \cdot \rangle
\]  
(3.27)

by T-Void. By applying the induction hypothesis on (3.27), we get:

★ If \( e_1 = E_1[e'] \), and \( e' \leadsto e'' \), then there exists \( E = \text{match} \ E_1 \ \text{with} \ \langle \rangle \) such that \( e = E[e'] \).

★ It is impossible for \( e_1 \) to be a value.

• **Case** \( e = \text{match} \ e_1 \ \text{with} \ \langle l \ x_1 \Rightarrow \text{rhs}_1 \ | \ x_2 \Rightarrow \text{rhs}_2 \rangle \)

We have

\[
\Delta, (\alpha_i : \kappa_i)_{i \in N}; \cdot \vdash e_1 : \langle l : t_1 \mid \rho \rangle
\]  
(3.28)

where \((\alpha_i)_{i \in N} = ftv(t_1) \cup ftv(\langle \rho \rangle)\) by T-Let. By applying the induction hypothesis on (3.28), we get:

★ If \( e_1 = E_1[e'] \), and \( e' \leadsto e'' \), then there exists \( E = \text{match} \ E_1 \ \text{with} \ \langle l \ x_1 \Rightarrow \text{rhs}_1 \ | \ x_2 \Rightarrow \text{rhs}_2 \rangle \) such that \( e = E[e'] \).

★ If \( e_1 \) is a value, it must be \((l' \ u)\) by Lemma 3.A.7, where \( l' \) may or may not equal \( l \), and then there exists a reduction with \( E = [] \) by ST-Match-Match and ST-Match-Skip, respectively.

• The rest of the cases can be routinely proven.

\(\square\)
Chapter 4

Traces for Safe Compositions

4.1 Introduction

In Chapter 3, we have seen the design of Typed ELEVATE, a row-polymorphic language for strategic rewriting, and its ability to express the strategies and strategy combinators in the original ELEVATE language [Hagedorn et al., 2020]. Typed ELEVATE provides useful features for strategy programming, such as precise structural types, recursive definitions, and pattern elaboration. Specifically, the extensible records and variants supported by the row polymorphism allow flexible encoding of ASTs, such as AST nodes with various numbers of sub-terms or ASTs with extra tags. On the other hand, there still exists problematic strategy compositions which definitely fail at runtime, but the causes of such unproductive compositions cannot be detected by Typed ELEVATE. In this chapter, we will discuss examples of these problematic strategies, and to statically detect their problems, we propose a type system, Rewrite $S_t$, with novel constructs called traces. We aim at designing a type system with traces for strategic rewriting, which preserves as many of the useful features of Typed ELEVATE as possible.

This chapter consists of the following contents:

- We discuss several examples showing the limitation of Typed ELEVATE in detecting problematic strategies which definitely fail at runtime (Section 4.2).
- We demonstrate the basic idea of traces with a graphical intuition (Section 4.3).
- We introduce the formalization of Rewrite $S_t$ in Section 4.4 and show the formal guarantees we can get in Section 4.5.
• We discuss the limitations of Rewrite $S_t$ in detail (Section 4.6) and conclude this chapter (Section 4.7).

4.2 Limitation of Typed ELEVATE

In this section, we will show some examples of problematic strategy compositions which definitely fail at runtime. For clarity, we use a heavily sugared syntax and fallible rewrite rule definitions in this section. For example, without syntactic sugar, the left-unit rule for multiplication, $\text{multUnitL}$, can be defined as follows in Typed ELEVATE.

\[
\text{-- } 1 \times n \rightarrow n \\
\text{let}\ \text{multUnitL} = \text{lam}\ expr = \text{match}\ expr\ \text{with} < \\
\ \\
\ \\
\text{App}\ \{\text{Fun}: \text{App}\ \{\text{Fun}: \text{BinOp}\ \text{Mul} | \text{Arg}: 1\} | \text{Arg}: n\} \rightarrow \text{Success}\ n \\
> \\
\text{multUnitL}\ \text{has the following type.} \\
\text{<App}: \{\text{Fun}: \text{<App}: \{\text{Fun}: \text{<BinOp}: \text{<Mul}: \{\} | \}* | \}* | \\
\text{Arg}: \text{<1}: \{\} | \}* | \} | \}* | \\
\text{Arg}: n | \} | \}* \\
\rightarrow \\
\text{<Success}: n | > \\
\]

In the LHS (Left-Hand Side) part of this type, all the rows in variant types are closed, which means they cannot be extended. We can use the $\text{multUnitL}$ rule in sequential compositions, such as $\text{seq} \ \text{multUnitL} \ \text{multUnitL}$. When $\text{multUnitL}$ gets an input of the wrong shape, such as $\text{seq} \ \text{mapFusion} \ \text{multUnitL}$, as expected, there will be a type error because the closed rows in the LHS type of $\text{multUnitL}$ cannot be unified with the rows in the RHS type of $\text{mapFusion}$. However, if $\text{multUnitL}$ is composed with $\text{mapFusion}$ using the $\text{lChoice}$ combinator, this apparently fine composition will result in a type error, because both $\text{multUnitL}$ and $\text{mapFusion}$ have closed rows in their LHS types, and their different LHS types cannot be unified when $\text{lChoice}$ expects both operands have the same LHS type.

To avoid similar problems, besides the $\text{Success}$ branch, a $\text{Failure}$ branch is added to the rewrite rule definition, where a variable pattern $\text{otherwise}$ is used to catch all expressions which cannot match the pattern of the $\text{Success}$ branch. Thus, we have a fallible definition of $\text{multUnitL}$ as follows.
let multUnitL = lam expr = match expr 
with < App {Fun: App {Fun: BinOp Mul | Arg: 1} | Arg: n} => Success n 
| otherwise => Failure

In the LHS type of this fallible multUnitL, the existence of the Failure branch makes all rows in the variant types extensible or open. Thus, for the composition mapFusion multUnitL, if mapFusion is also defined in a fallible way, we can typecheck their LHS types can be successfully unified, and this composition will typecheck.

However, if both multUnitL and mapFusion are defined in a fallible way, it is also possible to unify the LHS type of multUnitL with the RHS type of mapFusion in the sequential composition seq mapFusion multUnitL, which is rejected above when both operand rules are defined without the Failure branch. However, if both multUnitL and mapFusion are defined in a fallible way, their LHS types can be successfully unified and this composition will typecheck.

As the syntactic sugar, the form of the rule definitions above is too complex for presentation purposes. Thus, we write rule 1 * n -> n as a simpler form of the fallible multUnitL definition. In this section, all rewrite rules start with a rule keyword followed by the LHS, patterns and row types is also applied. Besides the seq combinator is written as an infix operator ; and the lChoice combinator is written as ||.

### 4.2.1 Checking Sequential Compositions

Starting with the sequential composition, let us consider a simple example rewriting arithmetic expressions: rule m * 0 -> 0; rule 1 * n -> n. Here we try to sequentially compose a rule that simplifies the multiplication with zero with the left-unit rule for multiplication. This composition must result in a Failure, as the expression returned by the first strategy is not compatible with the pattern expected by the second strategy.

However, with fallible rewrite rule definition, it is possible to unify the type of 0 (which is < 0 | >) and the type of 1 * n (which is < 1 * n | >), resulting in a type for presentation purposes. Thus, we write rule 1 * n -> n as a simpler form of the fallible multUnitL definition. In this section, all rewrite rules start with a rule keyword followed by the LHS, patterns and row types is also applied. Besides, the seq combinator is written as an infix operator ; and the lChoice combinator is written as ||.
Thus, in this case, Typed ELEVATE is not able to statically reject this erroneous composition.

### 4.2.2 Checking Left-choice Compositions

Can we check compositions involving the left choice combinator in the same way relying on type unification? To answer this, let us investigate another example where we either apply the left-unit rule for multiplication or the left-unit rule for addition:

\[
\text{rule } 1 \times n \rightarrow n \quad || \quad \text{rule } 0 + m \rightarrow m.
\]

To type this composition we must unify the LHSs of both strategy types (which are \(< 1 \times n | > \rightarrow < \text{Success: } n | \text{Failure } | >\) and \(< 0 + m | > \rightarrow < \text{Success: } m | \text{Failure } | >\)), as both could contribute to a successfully rewritten input – depending on if the first strategy was successfully applied or alternatively the first failed and the second was applied. The resulting unified type looks like \(< < 1 | 0 | > < * | + | > nm | >\).

As we represent programs with their AST and row-polymorphic type systems perform structural unification, both binary operations (+ and \(*\)) are unified into a single type, as are their arguments, making use of the implicit row variables. If this would be the type on the left-hand side of a strategy we would accept \(1 + z\) as a valid input even though neither strategy of the composition can handle it. This clearly shows that, row-polymorphic type unification in Typed ELEVATE is not sufficient for checking compositions involving the left choice combinator.

### 4.3 Rewrite \(S_t\), Intuitively

To address the limitations mentioned above, we propose a tracing system that is capable of distinguishing which parts of a unified type have been contributed by which strategy with special constructs called traces. This section will show the tracing mechanism with a graphical intuition.

#### 4.3.1 Traces for Checking Strategy Compositions

Since it is easier to show the intuition about the tracing mechanism with a left-choice composition, we start with the left-choice example from before. Let us examine the unified type annotated with traces and without the \text{Success} and \text{Failure} wrapper:
Chapter 4. Traces for Safe Compositions

The type is annotated with two traces, one in blue and one in red. In a structural type, a trace connects the parts belonging to the LHS of a strategy with the parts belonging to the RHS. In the figure here, these parts are marked by pins pointing out from each trace. Thus, a single trace shows one possible legal way that the strategy can transform its input to an output. Of course, a strategy can have multiple ways to transform its input. Therefore, the type of a strategy can be annotated with multiple traces each indicating a different possible execution path.

With traces representing the possible execution paths, we can reject inputs that do not follow any of the traces, such as the input \(1 + z\) mentioned before. We check this by typing and tracing the following sequential composition.

\[
\text{rule } ? \rightarrow 1 + z ; (\text{rule } 1 \times n \rightarrow n || \text{rule } 0 + m \rightarrow m)
\]

The ? means we do not care about the LHS of the first rule in this example as long as it produces \(1 + z\). The figure below shows the types and the corresponding traces of the two operands of the sequential composition. Here the yellow trace represents the execution path of \(\text{rule } ? \rightarrow 1 + z\):

To check if the traces connect, we unify the underlying RHS type of the first rule with the underlying LHS type of the left choice composition, and then the traces are aligned to check if they connect. To be aligned with the left choice composition with two traces, the trace of the first rule is duplicated.

As shown by the mismatched pins, the traces cannot be exactly aligned, indicating that no way of successful evaluation can be found. Such compositions of strategies are unproductive, and we can statically determine that their executions will always lead to the Failure case. If we insist on writing code which executes the composition, the type system with traces will reject it.
Now we switch to a composition where a legal execution path exists.

\[
\text{rule} \ ? \rightarrow \emptyset + z ; \ (\text{rule} \ 1 \ast n \rightarrow n \ || \ \text{rule} \ \emptyset + m \rightarrow m)
\]

The operands of this composition have the following types and traces.

\[
\begin{array}{c}
\text{?} \rightarrow < 0 | > < + | > z \\
\end{array}
\]

\[
\begin{array}{c}
< 1 | 0 | > < \ast | + | > \text{nm} \rightarrow \text{nm} \\
\end{array}
\]

As demonstrated above, we unify the types and align the traces, but this time, we can notice that one yellow trace perfectly aligns with the red traces representing \(\text{rule} \ \emptyset + m \rightarrow m\), indicating the existence of a legal execution path. Indeed, in this composition, \(\text{rule} \ ? \rightarrow \emptyset + z ; \ \text{rule} \ \emptyset + m \rightarrow m\) is a legal execution path.

With this intuition of traces, we can also see how it help us avoid sequential compositions which will definitely fail at runtime. Here we show the example

\[
\text{rule} \ m \ast \emptyset \rightarrow \emptyset ; \ \text{rule} \ 1 \ast n \rightarrow n
\]

mentioned above. At the beginning, the two rewrite rules have their own traces.

\[
\begin{array}{c}
m < \ast | > < 0 | > \rightarrow < 0 | > \\
\end{array}
\]

\[
\begin{array}{c}
< 1 | > < \ast | > \text{n} \rightarrow \text{n} \\
\end{array}
\]

Then we unify the RHS type of the first rule with the LHS type of the second one while maintaining the traces, and check if the the traces align with each other.

\[
\begin{array}{c}
m < \ast | > < 0 | > \rightarrow < 0 | > < 1 | > < \ast | > \text{n} \rightarrow \text{n} \\
\end{array}
\]

In this figure, we can see that all the pins do not align, meaning the pattern matching will absolutely fail. In our type system with traces, the execution of such composition will be rejected.
4.4 Rewrite $S_I$, Formalized

In this section, we present the core calculus Rewrite $S_I$ which formalizes our intuition of traces. We show the syntax of terms and types, the semantics and the type system leveraging row-polymorphism. The meta-level syntax conventions used in this chapter are the same as those in Chapter 3 (see Section 3.3).

4.4.1 Term Syntax

Figure 4.1 shows the syntax of terms in Rewrite $S_I$. Terms (denoted by $e$) share some common constructs with Typed ELEVATE, including variables ranging over $x, y, z, \ldots$, term applications ($e_1 e_2$), lambda abstractions ($\text{lam } x \rightarrow e$), let-bindings ($\text{let } f = e_1 \text{ in } e_2$), field accesses ($e.l$), field removals ($e.-l$), record modifications ($e.\{ \} \in \mathbb{N}$), record modifications ($e.\{ \} \in \mathbb{N}$), record extensions ($e.+\{ \} \in \mathbb{N}$), and pattern matching with the empty pattern ($\text{match } e \text{ with } (\{\} \rightarrow e_1)$), the unit (empty record) pattern ($\text{match } e \text{ with } (\{\} \rightarrow e_1)$) or the variant pattern ($\text{match } e \text{ with } (\{\} \rightarrow e_1)$). The terms above are mainly for general-purpose programming and the remainder parts are strategy-related language constructs. Rewrite rules are defined with lambda-like syntax starting with the keyword $\text{rule}$. A rewrite rule can accept any argument ($\text{rule } x \rightarrow e$), match the argument with a variant pattern ($\text{rule } l x \rightarrow e$), match the argument with a record pattern ($\text{rule } l : x | r \rightarrow e$), or match the argument with the unit pattern ($\text{rule } \{\} \rightarrow e$). To simplify the tracing of rewrite rules, which will be introduced later, record operations except record construction are not allowed in rewrite rules, so the record pattern ($\{ l : x | r \}$) is used to deconstruct record by introducing a variable $x$ for the field labelled by $l$, and a variable $r$ for the rest of the record.
In the body of a rewrite rule, to construct a complex pattern with simple ones like the elaborated patterns in Typed ELEVATE, a dedicated rule application $x \rightsquigarrow e$ is provided to create cascaded pattern matchings: it takes a variable $x$ bound by a previous pattern, and supply it to the next rule $e$ as the argument. Returning the rewrite result also has its own syntax ($\text{return } e_1 e_2$) where $e_2$ is the expected result and $e_1$ computes a Boolean predicate to decide whether the rewrite is successful, i.e., whether $e_2$ can be returned. To compose rewrite rules or strategies, we need strategy combinators which can be constructed with the strategy combinator lambda ($\text{st } x \Rightarrow e$) and the strategy combinator application ($e_1 \leftarrow e_2$). Finally, a fully applied strategy $e$ can be evaluated with $\text{run } e$.

**Constructing a Rewrite Rule**  As mentioned above, the construction of complex patterns in Rewrite $S_t$ resembles the elaborated patterns in Typed ELEVATE. To show this feature clearly, the formal representation of rule $1 * n \rightarrow n$ is shown below.

\[
\begin{align*}
\text{rule } & \text{App } f x_0 \rightarrow f x_0 \leftarrow \\
\text{rule } & \{ \text{Fun} : f_0 | r_0 \} \rightarrow f_0 \leftarrow \\
\text{rule } & \text{App } f x_1 \rightarrow f x_1 \leftarrow \\
\text{rule } & \{ \text{Fun} : f_1 | r_1 \} \rightarrow f_1 \leftarrow \\
\text{rule } & \text{Mul } u_0 \rightarrow u_0 \leftarrow \text{rule } \{ \} \rightarrow r_1 \leftarrow \\
\text{rule } & \{ \text{Arg} : x_1 | r_2 \} \rightarrow x_1 \leftarrow \\
\text{rule } & 1 u_1 \rightarrow u_1 \leftarrow \text{rule } \{ \} \rightarrow r_2 \leftarrow \\
\text{rule } & \{ \} \rightarrow r_0 \leftarrow \\
\text{rule } & \{ \text{Arg} : n | r_3 \} \rightarrow r_3 \leftarrow \\
\text{rule } & \{ \} \rightarrow \text{return } (\text{True } \{ \}) n
\end{align*}
\]

In this long chain of rewrite rules, the bound variables from previous rules are matched, or decomposed by subsequent rules, describing the complex pattern below.

\[
\text{App } \{ \text{Fun} : \text{App } \{ \text{Fun} : \text{Mul} | \text{Arg} : 1 \} | \text{Arg} : n \}
\]

**4.4.2 Kinds and Types**

Figure 4.2 shows the syntax of kinds, traces and types in Rewrite $S_t$, together with the syntax of the kinding, tracing and typing environments. The usage of row polymorphic types in Rewrite $S_t$ is inspired by Blume et al. [2006]. Types are of two main
### Kinds
\[ \kappa :\ T\ |\ T_\bot\ |\ T_0\ |\ T_\star\ |\ R \]
\[ R :\ L\ |\ \neg L \]
\[ L :\ \{(l_i)_{i\in\mathbb{N}}\} \]

### Traces
\[ \varphi :\ \alpha\ |\ \phi \]

### Bunches
\[ \phi :\ \{(\varphi_i)_{i\in\mathbb{N}}\} \]

### Tracing Status
\[ \upsilon :\ \Box\ |\ \phi \]

### Types
\[ t :\ \alpha\ |\ \tau\ |\ \omega\ |\ t_1 \rightarrow\ t_2 \]
\[ \tau :\ \alpha[\phi]\ |\ \{\rho\}\ |\ \langle\rho\rangle \]
\[ \rho :\ \alpha[\phi]\ |\ \langle\phi\rangle\ |\ l[\phi] :\ \tau\ |\ \rho \]
\[ \omega :\ \tau_1 \phi\rightarrow\tau_2\ |\ (\tau_1 \phi\rightarrow\tau_2) \Rightarrow\omega \]
\[ s :\ \tau\phi\ |\ \phi\tau \]

### Schemes
\[ \sigma :\ \text{tr}\ \Phi,\ \overline{\sigma}\ |\ \overline{\sigma} :\ \forall\ (\alpha :\ \kappa).\ \overline{\sigma}\ |\ t \]

### Kinding Env.
\[ \Delta :\ \cdot\ |\ \Delta,\ \alpha :\ \kappa \]

### Strategy Combinator Env.
\[ \Sigma :\ \cdot\ |\ \Sigma,\ x :\ \omega \]

### Typing Env.
\[ \Gamma :\ \cdot\ |\ \Gamma,\ x :\ \sigma \]

### Partial Tracing Env.
\[ \Phi :\ \cdot\ |\ \Phi,\ \alpha :\ \phi \]

### Rule Env.
\[ \Theta :\ \cdot\ |\ \Theta,\ x :\ s\ |\ \Theta,\ \ast :\ \alpha \]

### Tracing Env.
\[ \Phi :\ \phi\triangleright\overline{\Phi} \]

---

Figure 4.2: Syntax of kinds, traces and types (top) and kinding, tracing and typing environments (bottom).
categories of kinds: $\mathcal{T}$ for ordinary types, and $\mathcal{R}$ for row types. $\mathcal{T}$ can be further divided into three subkinds: $\mathcal{T}_{-}$ for types which will not be traced, $\mathcal{T}_{\circ}$ for types which can be traced, and $\mathcal{T}_{\bullet}$ for types which are already traced and packed. Further details will be given in Section 4.4.3.

A novel and important part of our type system is tracing. Metaphorically, traces can be imagined as threads of different colours each marking a possible way of rewriting. Traces ($\varphi$) can be trace variables ($\alpha$) or non-repetitive collections of traces, called bunches ($\phi$). Using the thread metaphor, each trace is a thread connecting trace variables, and one of the trace variables (the "parent" trace variable) is so important that it could break the whole thread, therefore it can be used to identify the entire trace. The rest of the trace variables (the "child" trace variables) can be replaced or removed without breaking the thread. However, removing the child trace variables causes accumulative damages to the thread, and when all child trace variables are removed, the thread also breaks. Besides, as its name suggests, the tracing status ($\nu$) indicates the status of tracing a type: untraced ($\Box$) or traced by a bunch $\phi$.

Ordinary types ($t$) include type variables ($\alpha$), traceable types ($\tau$), strategy combinator types ($\omega$) and function types ($t_1 \to t_2$). The traceable types contain traceable type variables, the record types ($\{\rho\}$) and variant types ($\langle\rho\rangle$). When a type variable is traced, a bunch containing the traces will appear at the subscript position.

Row types ($\rho$) are sequences of label-type pairs $l : t$ ending with row variables or empty rows ($\cdot$), where the order of the label-term pairs is insignificant, and labels are all distinct. The formalization of row types in this chapter is the same as that in Chapter 3 (see Section 3.3).

A first-order strategy combinator type, or strategy type, $\tau_1 \overset{\phi}{\rightarrow} \tau_2$ represents a strategy from $\tau_1$ to $\tau_2$, traced by $\phi$, and $\phi$ collects all parent trace variables from the traces in $\tau_1$ and $\tau_2$. In other words, the number of trace variables in $\phi$ represents the number of possible ways of rewriting from $\tau_1$ to $\tau_2$. Thus, we call $\phi$ the parent bunch of $\tau_1$ and $\tau_2$. A second-order strategy combinator type, or strategy combinator type, $(\tau_1 \overset{\phi}{\rightarrow} \tau_2) \Rightarrow \omega$ represents a strategy combinator which takes a strategy $\tau_1 \overset{\phi}{\rightarrow} \tau_2$ as its parameter and produces the resulting strategy (combinator) $\omega$. As justified later in Section 4.6, if a strategy combinator is treated as a function of strategies, it is linear and the parameter strategies must be used exactly once.

There is a special group of types ($s$) solely used in rewrite rule definitions, and they will not appear in other places like type schemes, so they are not included as part of the definition of $t$. Type $s$ includes rule argument type ($\tau^{\phi}$) and rule return type
Like the parent bunches in strategy combinator types, the $\phi$ here also provides collective tracing information.

A type schemes ($\sigma$) represents possibly universally quantified types, and it may also carry a tracing environment, to be instantiated with fresh traces. Type schemes without tracing environment are denoted as $\overline{\sigma}$. The kind of the bound type variables are specified at the binding site.

Figure 4.2 also gives the definitions of various environments. The kinding ($\Delta$) and typing environments ($\Gamma$) are standard: they can be either empty ($\cdot$) or extended with a type variable and its kind ($\alpha : \kappa$) or a variable and its type scheme ($x : \sigma$). The rule environment ($\Theta$) is the typing environment used while defining a rewrite rule. A rule environment only contains variables of rule argument type, or (possibly repetitive) nameless placeholders ($\star$) which make the rule environment non-empty without introducing actual variables. Besides, each $\star$ is associated with a trace variable to maintain the tracing information in the absence of an actual variable-type pair. The tracing environment ($\Phi$) consists of two parts: a partial tracing environment ($\overline{\Phi}$) and a parent bunch $\phi$. Each element in the partial tracing environment associates a trace variable $\alpha$ with a trace $\varphi$ whose content must be found in $\phi$, that is, the parent-child hierarchy should be made explicit by the tracing environment. This well-formedness condition is formalized in Figure 4.12 and Figure 4.13 in Section 4.A.3, and we always assume a tracing environment is well-formed. The strategy combinator environment only holds variables of the strategy combinator type. As part of the linear typing of strategy combinators, the strategy combinator environment is linear.

Connecting the Formalism With the Graphical Intuition  Section 4.3 shows the graphical intuition for traces using colourful lines attached to types, and this intuition can be encoded in the formal syntax above.

For example, the first graph in Section 4.3 is as follows.

```
< 1 | 0 |> < * | + |> nm $\rightarrow$ nm
```

If the underlying row polymorphic types (except the spaces representing function applications) are desugared, the graph will be:

```
< 1: \{\cdot\} | 0: \{\cdot\} | rx > < *: \{\cdot\} | +: \{\cdot\} | ry > nm $\rightarrow$ nm
```
As explained in Section 4.3, a trace marks the LHS and the RHS of a possible execution path with pins, and formally these pins are encoded as trace variables, as shown by the figure below.

In this example, the blue trace is identified by the parent trace variable \( a \), and the red one is identified by \( b \). It is worth noting that in both traces, the type variable \( nm \) is traced by child trace variables (\( a0 \) and \( b0 \), respectively) on both sides of the strategy type. With these trace variables, we can write this example in an "almost" formal syntax as follows. For the consistency with other formal presentation of row types in this thesis, the \( * \) and \( + \) operators are replaced by \( \text{Mul} \) and \( \text{Add} \), respectively.

\[
\langle 1[a] : \{ \cdot \} | 0[b] : \{ \cdot \} | rx \rangle \langle \text{Mul}[a] : \{ \cdot \} | Add[b] : \{ \cdot \} | ry \rangle \xrightarrow{[a,b]} nm[a0,b0]
\]

According to the typing rules to be introduced in the next section, the unit types and row variables should also have tracing information, but these added trace variables would not affect the overall structure of traces. Besides, the row variables \( rx \) and \( ry \) do not correspond to any execution paths, so their traces are empty.

\[
\langle 1[a] : \{ [a1] \} | 0[b] : \{ [b1] \} | rx[1] \rangle \langle \text{Mul}[a] : \{ [a2] \} | Add[b] : \{ [b2] \} | ry[1] \rangle \xrightarrow{[a,b]} nm[a0,b0]
\]

To completely desugar this type, we replace the spaces with actual function application types and then get the strategy type below. Since the nested function applications are shared by both execution paths, the relevant labels and unit types are traced by traces identified by both \( a \) and \( b \).

\[
\langle \text{App}[a,b] : \{ \text{Fun}[a,b] : \rangle \langle \text{App}[a,b] : \{ \text{Fun}[a,b] : \langle \text{Mul}[a] : \{ [a2] \} | Add[b] : \{ [b2] \} | ry[1] \rangle | \rangle | Arg[a,b] : \langle 1[a] : \{ [a1] \} | 0[b] : \{ [b1] \} | rx[1] \rangle | [a3,b3] \rangle | rz[1] \rangle | \xrightarrow{[a,b]} nm[a0,b0] \rangle
\]
If this strategy is let-bound and assigned a type scheme, the tracing environment and the kinds of row variables will be included in the type scheme as follows, and this strategy can be used polymorphically in different expressions.

\[ \text{tr} \ [a, b] \triangleright (a_0 : a), (a_1 : a), (a_2 : a), (a_3 : a), (a_4 : a), (b_0 : b), (b_1 : b), (b_2 : b), (b_3 : b), (b_4 : b). \]

\[ \forall (rr : \neg \{\text{App}\}). \forall (rx : \neg \{1, 0\}). \forall (ry : \neg \{\ast, +\}). \forall (rz : \neg \{\text{App}\}). \]

\[
\langle \text{App}_{[a,b]} : \{\text{Fun}_{[a,b]} : \langle \text{Mul}_{[a]} : \{\cdot_{[a_1]}\} | \text{Add}_{[b]} : \{\cdot_{[b_1]}\} | ry[r] \rangle | \\
\text{Arg}_{[a,b]} : \langle 1_{[a]} : \{\cdot_{[a_1]}\} | 0_{[b]} : \{\cdot_{[b_1]}\} | rx[r] | \cdot_{[a_3,b_3]} | rz[r] \rangle | \\
\text{Arg}_{[a,b]} : \text{nm}_{[a_0,b_0]} | \cdot_{[a_4,b_4]} | rz[r] \rangle 
\]

\[ \crule{[a,b]}{nm_{[a_0,b_0]}} \]

### 4.4.3 Type System

**Traces and Patterns**

The general idea of *tracing* is to maintain the information on the pattern matching performed by a strategy in its type. Given the type \( \tau_1 \rightarrow \langle \text{Success} : \tau_2 | \text{Failure} : \{\cdot\} | \cdot \rangle \) of a strategy as an ordinary function without any tracing information, we could not tell how this strategy is constructed: it may just be a single rewrite rule; it may also be the sequential or parallel composition of multiple strategies and what we see are only types after unification. Thus, we could not *statically* determine if the strategy would fail given an argument of type \( \tau_1 \) because of the inaccessibility of the exact pattern matching process inside the strategy. To solve this problem, traces are introduced to keep track of the pattern information.

According to the syntax of traceable types and patterns for rule definitions (with the *rule* keyword), traceable types can perfectly represent patterns, either used as strategy arguments or return values. The most significant part of a refutable pattern is the *label*: the mismatch of labels in variant patterns or the absence of expected labels in record patterns would make the whole pattern matching fail. Thus, labels which actually appear in patterns are assigned the parent trace variable. As for pattern variables, which can be variable patterns or the variables in variant patterns, they can be refined during rewriting, so when we look at the types, the parts corresponding to pattern variables may be variant or record types containing labels instead of solely type variables. However, failing to match (some of) these labels will not cause the
failure of the whole pattern matching. Thus, pattern variables are assigned the child trace variables according to the description in Section 4.4.2. A small example is the following:

\[
\langle A_{[x,y_0]} : \{x_0,y_1\} \mid B_{[y_2]} : \{y_3\} \mid r_{[y_4]} \rangle
\]

In this example, trace variables with a numeric subscript are child trace variables. By noticing that the parent trace variable \(x\) only appears in the case of label \(A\) and the whole type is a variant, this example could represent the left-choice composition of the variant pattern \(A_p\) and the variable pattern \(q\) whose type is refined to \(\langle A : \{\cdot\} \mid B : \{\cdot\} \mid r \rangle\). If a value of type \(\langle B : \{\cdot\} \mid r \rangle\) is matched against each pattern, it is obvious that the variant pattern \(A_p\) will fail and the whole trace related with \(x\) will disappear.

On the other hand, for the variable pattern \(q\), it fails at the case of label \(A\) but succeeds at the case of label \(B\), and only the trace variables \(y_0\) and \(y_1\) will be removed.

For convenience, the following automatic conversion rules are used throughout this chapter. Nested bunches will be concatenated as needed. Appending a tracing environment to another will result in combining the parent bunches and partial tracing environment.

\[
[[((\varphi_i)_{i \in N}, (\varphi_j)_{j \in M}) = [(\varphi_i)_{i \in N}, (\varphi_j)_{j \in M}] \quad (\phi_1 \triangleright \Phi_1, \phi_2 \triangleright \Phi_2) = [\phi_1, \phi_2] \triangleright (\Phi_1, \Phi_2)
\]

Kinding Rules

Figure 4.15 in Section 4.A.3 gives the kinding rules, which also work as the well-formedness rules for types. Kinding judgments are of the form \(\Delta; \Phi \vdash t : \kappa \triangleright \nu\), stating that 'the type \(t\) has kind \(\kappa\) (and is well formed) in tracing environment \(\Phi\) and the kinding environment \(\Delta\), and the tracing status of \(t\) is \(\nu\) which contains the parent bunch of \(t\) if \(t\) is traced'. Some of the kinding rules, and some of the typing rules in the subsequent text, use the row kind subset relation (\(\subseteq\)) and the row kind extension operator (+), which are inherited from Chapter 3 (see Figure 3.4).

Most of the kinding rules are straightforward, and a representative one is shown below.

\[
\Delta; \Phi \vdash \rho : \mathcal{R} \triangleright \phi^\rho \quad \Delta; \Phi \vdash \tau : \mathcal{T}_\rho \triangleright \phi^\tau
\]

\[
\phi = \phi_\rho + \phi^\tau + \text{parent}(\phi_l, \Phi) \quad \mathcal{R} + l = \mathcal{R}^\text{ext}
\]

\[
\Delta; \Phi \vdash (l_{\phi_l} : \tau \mid \rho) : \mathcal{R}^\text{ext} \triangleright \phi \quad \text{K-RowExt-Traced}
\]

This **K-RowExt-Traced** rule shows the usage of the row kind extension operator: it extends the kind with a label and rules out rows containing repeated labels. It also
shows how the parent bunch of the row is computed. The definition of the (over-
loaded) + operator for bunches and the parent function can be found in Figure 4.12
in Section 4.A.3. The + operator connects bunches together while ignoring repeated
trace variables. The parent function computes the parent bunch of a bunch given the
tracing environment.

A special kinding rule is \textbf{K-TVar-Traced} as follows.

$$
\alpha : \kappa \in \Delta \quad \phi_1 = [(a_i)_{i \in N}] \quad \Phi = \phi \triangleright (a_i : a'_i)_{i \in N}, \overline{\Phi}
$$

Besides extracting the parent bunch of $\phi_1$, it checks if all traces in $\phi_1$ are child trace
variables, because a type variable can only be part of the type of a pattern variable.

To avoid tracing non-traceable types such as ordinary function types, a subkind
relation ($<$) is defined in Figure 4.15. There is a chain $\mathcal{T}_\varnothing < \mathcal{T}_\mathcal{N} < \mathcal{N}$ in this relation,
and $\mathcal{T}_\varnothing$ does not have any relation with other kinds except being the subkind of $\mathcal{N}$. It
should be noted that although strategy combinator types contain traced types, them-
selves cannot be traced again. The subkind relation also prevents strategy combinator
types from being used as the parameters of ordinary functions (see \textbf{K-Func-Traced}),
otherwise the linearity will be violated.

\section*{Trace Computation}

When operating with traces, e.g. when connecting them, we need to perform com-
cputations with traces. In this part, we show some important definitions for trace
computation. The multiplication of traces is used to compute all possible connec-
tions of execution paths. It is defined below, and using the thread metaphor will help
us explain how it works. Given a bunch of threads with $n$ ends, and a bunch with $m$
ends, we will need $n \times m$ new threads to enumerate all possible ways to connect this
two bunches. The $\times$ operator used in the matrix is overloaded so that a fresh trace
variable $\alpha''$ is created when $\alpha \times \alpha'$. The triplet returned by the trace multiplication
gives us the bunch of new threads, and how the new threads can be connected to each
input bunch, respectively.

$$
N = [1 \ldots n] \quad M = [1 \ldots m] \quad \forall i \in N, S_i = (a_i \mapsto \{\text{row}_i(P)\})
$$

$$
P = \begin{cases}
(a_1 \times a'_1) & \cdots & (a_1 \times a'_m) \\
\vdots & \ddots & \vdots \\
(a_n \times a'_1) & \cdots & (a_n \times a'_m)
\end{cases}
\quad \forall j \in M, S'_j = (a'_j \mapsto \{\text{col}_j(P)\})
$$

$$
S = (S_i \circ)_{i \in N}() \quad S' = (S'_j \circ)_{j \in M}()
$$

$$
([\text{vec}(P)], S, S') = [(a_i)_{i \in N}] \times [(a'_j)_{j \in M}]
$$
Chapter 4. Traces for Safe Compositions

Reusing one of the examples from Section 4.3, the multiplication of traces is visualised as follows. The purple and green dashed lines represent the two newly created trace variables, respectively.

For a more intuitive presentation, Section 4.3 does not precisely show the exact type checking process and omits some internal details which will be revealed here. If the substitutions returned by the trace multiplication are applied, we get the following result. The new trace variables replace the old ones, so the colours of the lines change correspondingly. It is worth noting that the trace variable identifying the yellow trace is substituted by a bunch containing the trace variables identifying the purple and green traces. In the graph, this is shown as adjacent traces on the left.

With the thread metaphor, trace operations like trace multiplication expect each trace in the input bunches to be a single thread end-to-end, but as shown by the graphs above, we can easily create a trace containing multiple threads by trace multiplication. To maintain this single thread condition in the tracing environment, we say a tracing environment is normalized when each child trace variable is only associated with one parent trace variable. To perform this operation, $\text{norm}$ is defined in Figure 4.18 in Section 4.A.3. For each child trace variable in $\Phi$, $\text{norm}(\Phi)$ creates fresh child trace variables for each trace variable in its associated parent bunch, and removes the child trace variable if it is associated with an empty bunch. The graph below shows the result of normalization.

While typing checking strategy compositions, the unification of traced types is an important operation. Since we cannot tell the difference between two empty bunches, it is always possible to make two traces equal by substituting all trace variables with the empty bunch, but this makes no sense when we want the traces to carry pattern information. Thus, to avoid similar nonsensical substitutions, the unification of
traced types is performed during type checking to make sure that the traces faithfully represent the patterns. Figure 4.3 defines the unification of traces. Figure 4.17 in Section 4.A.3 gives the definition of the unification of bunches and traced types, which are simple enumeration of traces and trivial traversal of types hence not shown here. For the `unifyTrace` definition, the thread metaphor still works here: two threads can only be merged if they have the same color. The unification process follows these rules: a trace variable always unifies with itself; a child trace variable unifies with its parent; a child trace variable unifies with another child trace variable if they have the same parent; otherwise the unification fails. The graph below shows the result of traced type unification, where the black crosses mark failed trace unifications.

Besides, there is another operation, `distr`, defined in Figure 4.16 in Section 4.A.3. It distributes fresh trace variables over the whole input untraced type, returns the new traced type and a bunch containing all freshly generated trace variables. A specific use case is that if all the trace variables generated by `distr(\(\tau\))` are defined as child trace variables afterwards, `distr(\(\tau\))` produces a traced type associated with a pattern variable.
Typing Judgements and Typing Environments

Typing judgments appear in the form $\Delta; \Gamma; \Phi; \Sigma; \Theta \vdash e : t \rightarrow \Phi'; \Sigma'; \Theta'$, stating that ‘the types in the typing environment $\Gamma$ are well formed in the kinding environment $\Delta$; the types in the strategy combinator environment $\Sigma$ and the rule environment $\Theta$ are well formed in $\Delta$ and the tracing environment $\Phi$; the term $e$ has type $t$ in $\Phi'$, $\Delta$, $\Gamma$, $\Sigma'$ and $\Theta'$; finally, $t$, $\Sigma'$ and $\Theta'$ are well formed in $\Delta$ and $\Phi'$. The equivalence of types is up to alpha-renaming. All the environments at the left of the turnstile are call input environments, and all the environments at the right of the reverse turnstile are called the output environments. If any of the environment must be empty, we omit it from the typing judgement. For example, $\Delta; \Gamma; \Phi; \Sigma; \Theta \vdash e : t \rightarrow \Phi'; \Sigma'; \Theta$. If the output environments are exactly the same as their input counterparts, we omit them together with the reverse turnstile. For example, $\Delta; \Gamma; \Phi; \Theta \vdash e : t$ is equivalent to $\Delta; \Gamma; \Phi; \Theta \vdash e : t \rightarrow \Phi; \Sigma; \Theta$. Besides simply calling them input or output environments, in a type judgement $\Delta; \Gamma; \Phi; \Sigma; \Theta \vdash e : t \rightarrow \Phi'; \Sigma'; \Theta'$, we say $\Phi$ is the raw tracing environment, and $\Phi'$ the refined tracing environment. The same convention applies to $\Sigma$ and $\Sigma'$, $\Theta$ and $\Theta'$, respectively. The reason for this naming will be explained later. The notation $\text{fto}(t)$ or $\text{fto}(\Gamma)$ stand for the set of free type variables in type $t$ or typing environment $\Gamma$, respectively.

Shown below are two built-in items in the typing environment $\Gamma$ by default. They are the sequential combinator $\text{seq}$ and the left-biased choice combinator $\text{lChoice}$.

\[
\text{seq} : \text{tr} [a] \triangleright (a_0 : a), (a_1 : a), (a_2 : a). \forall (r : T_0) (s : T_0) (t : T_0).
\]

\[
(r[a_0] \xrightarrow[a]{} s[a_1]) \Rightarrow (s[a_1] \xrightarrow[a]{} t[a_2]) \Rightarrow (r[a_0] \xrightarrow[a]{} t[a_2])
\]

\[
\text{lChoice} : \text{tr} [a, b] \triangleright (a_0 : a), (a_1 : a), (b_0 : b), (b_1 : b). \forall (p : T_0) (q : T_0).
\]

\[
(p[a_0] \xrightarrow[a]{} q[a_1]) \Rightarrow (p[b_0] \xrightarrow[b]{} q[b_1]) \Rightarrow (p[a_0, b_0] \xrightarrow[a, b]{} q[a_1, b_1])
\]

The underlying types without traces here resemble the type of the corresponding combinators in Typed ELEVATE, while traces provide extra information about the behaviour of the combinators. The type of $\text{seq}$ only contains one trace as sequential composition itself does not introduce any branching structure. The type of $\text{lChoice}$, on the other hand, puts two traces from the input strategies together as the left-biased choice composition has two branches or execution paths.
Typing Rules for Untraced Terms

The typing rules for the $\lambda$-calculus subset of Rewrite $S_t$ (variables, term applications, lambda abstractions) are almost standard. The corresponding typing rules are listed in Figure 4.7 in Section 4.A.1. It should be pointed out that all typing rules in Figure 4.7 have empty $\Sigma$ and $\Theta$, preventing the corresponding terms from being used within any tracing-related language constructs.

Typing Rules for Rewrite Rules

Figure 4.4 shows some representative typing rules for elementary strategies, that is, rewrite rules. The complete set of typing rules is in Figure 4.19 in Section 4.A.3. To simplify type checking for rewrite rules, some restrictions are added: a rewrite rule can only have a single trace; no variables from $\Gamma$ or $\Sigma$ can be used within a rewrite rule except in the predicate part of return; the traces of rule argument types can only be modified by the patterns following the rule keyword. Thus, for all typing rules in Figure 4.19, the strategy combinator environment $\Sigma$ is empty, and only basic variant and record constructions are allowed because operations like field access can change the traces by requiring the existence of a specific label.

The typing rule T-R-Lam-Label is for rewrite rule definitions with variant pattern, and it is a representative example to show the tracing mechanism. In the conclusion of this rule, being similar to the T-Match rule from Typed ELEVATE, the type $\langle l_{\alpha} : \tau_x | \rho \rangle$ corresponds to the variant pattern $l x$. To start typing this rewrite rule definition, we first check if the raw tracing environment $\Phi$ is empty. If so, it means this rewrite rule is not inside the body of any other rewrite rules, so we must create a fresh parent trace variable $\alpha$ for subsequent tracing; if not, there must be another rewrite rule creating the parent trace variable, and in this rule, we must reuse it to ensure that there is only a single trace. With the parent trace variable, we can already trace the label $l$. To trace the variable $x$, we extract the underlying untraced type of $\tau_x$ with the erase function defined in Figure 4.14 in Section 4.A.3, and then distribute child trace variables over the erased type so that the trace in $\tau_{new}$ is known to represent a pattern variable. $\tau_{new}[\alpha]$ is then put into the raw rule environment $(\Theta, x : \tau_{new}[\alpha])$ for type checking the rule body $e$ where the variable $x$ may be further analysed by other rewrite rules, and finally we get the expected type back in the refined rule environment $(\Theta', x : \tau_{x}[\alpha])$. Besides, since we know that any input that does not match the pattern $l x$ would make the rewrite rule fail, so $\rho$ must be completely traced by the empty bunch. This
If $\Phi = [] \triangleright \cdot$ Then fresh $\alpha$ $\overline{\Phi} = \cdot$ Else $\Phi = [\alpha] \triangleright \overline{\Phi}$

$([[\alpha_i]_{i \in N}, \tau_{new}) = \text{distr}(\text{erase}(\tau_x)) \quad \Phi_{ext} = [\alpha] \triangleright \overline{\Phi}, (\alpha_i : \alpha)_{i \in N}$

$\rho = \text{clear}(\rho) \quad \Delta; \Gamma; \Phi_{ext}; \Theta, x : \tau_{new} [\alpha] \vdash e : [\alpha] \tau_e \rightarrow \Phi'; \Theta', x : \tau_x [\alpha]$  \[\text{T-R-Lam-Label}\]

$\Delta; \Gamma; \Phi; \Theta \vdash \text{rule } l \ x \rightarrow e : \langle l_{[\alpha]} : \tau_x \mid \rho \rangle \rightarrow [\alpha] \tau_e + \Phi'; \Theta'$

$\rho = \text{clear}(\rho) \quad \Delta; \Gamma; \Phi_{ext}; \Theta, x : \tau_{new} [\alpha] \vdash e : [\alpha] \tau_e \rightarrow \Phi'; \Theta'$

$(\Phi', S) = \text{unify}(\Phi', \tau, \tau_1) \quad \tau'_2 = \tau_2 [S] \quad \tau' = \tau [S]$  \[\text{T-R-App}\]

$\Delta; \Gamma; \Phi; \Theta \vdash x \leftarrow f : [\alpha] \tau'_2 + \Phi'' ; \Theta', x : \tau' [\alpha]$  

$\Theta \neq \cdot \quad \Delta; \Gamma; \Phi; \Theta \vdash e_2 : \tau^{[\alpha]} + \Phi'$

$\Delta; \Gamma, \text{erase} (\Theta) \vdash e_1 : \langle \text{True} : \{\cdot\} \mid \text{False} : \{\cdot\} \mid \cdot \rangle$  \[\text{T-R-Return}\]

$\Delta; \Gamma; \Phi; \Theta \vdash \text{return } e_1 e_2 : [\alpha] \tau + \Phi'$

Figure 4.4: Representative Typing rules for rewrite rules.

is achieved by using the clear function defined in Figure 4.14 in Section 4.A.3.

In T-R-Lam-Label, types with freshly generated (raw) traces are put into the raw rule environment, and the refined rule environment holds types with potentially refined traces. The typing rule T-R-App is where the refinement happens: the variable $x$ is introduced by some previous patterns, and its type $\tau$ is refined by unifying with the parameter type $\tau_1$ of $f$, whose trace represent a pattern defined by $f$. It should be noted that $(x : \tau^{[\alpha]}$) is taken from the refined rule environment of the typing judgment for $f$ because the type of $x$ may be further refined inside $f$.

Most of the remaining typing rules in Figure 4.19 are defined following the similar mechanism as T-R-Lam-Label. Notably, in T-R-Lam-Unit shown below with irrelevant premises hidden, the ★ placeholder is used to make the rule environment non-empty without introducing a variable, so any subsequent language constructs would know they are in a rewrite rule definition.

$\Delta; \Gamma; \Phi_{ext}; \Theta, \star : \alpha_1 \vdash e : [\alpha] \tau_e + \Phi'; \Theta', \star : \alpha_1$  \[\text{T-R-Lam-Unit}\]

$\Delta; \Gamma; \Phi; \Theta \vdash \text{rule } \{\} \rightarrow e : \{[\alpha_1] \} \rightarrow [\alpha] \tau_e + \Phi'; \Theta'$

The rule T-R-Return converts the rule argument type $\tau^{[\alpha]}$ into the rule return type...
\[ \Delta; \Gamma; \Phi_1; \Sigma_1 \vdash f : (\tau_1 \rightarrow \tau_2) \Rightarrow \omega' + \Phi'_1; \Sigma'_1 \]
\[ \Delta; \Gamma; \Phi_2; \Sigma_2 \vdash e : \tau_3 \rightarrow \tau_4 + \Phi'_2; \Sigma'_2 \]
\[ (\phi_3, S_{1 \rightarrow 3}, S_{2 \rightarrow 3}) = \phi_1 \times \phi_2 \quad (\phi'_1 \triangleright \Phi_{n1}, S_{n1}) = \text{norm}(\Phi'_1[S_{1 \rightarrow 3}]) \]
\[ (\phi_3 \triangleright \Phi_{n2}, S_{n2}) = \text{norm}(\Phi'_2[S_{2 \rightarrow 3}]) \quad \Phi' = \phi'_1 \triangleright \Phi_{n1}, \Phi_{n2} \]
\[ S_{n1 \rightarrow 3} = S_{n1} \circ S_{1 \rightarrow 3} \quad S_{n2 \rightarrow 3} = S_{n2} \circ S_{2 \rightarrow 3} \]
\[ (\Phi_{u1}, S_{u1}) = \text{unify}(\Phi', \tau_1[S_{n1 \rightarrow 3}], \tau_3[S_{n2 \rightarrow 3}]) \]
\[ (\Phi_{u2}, S_{u2}) = \text{unify}(\Phi_{u1}, \tau_2[S_{n1} \circ S_{n1 \rightarrow 3}], \tau_4[S_{n1} \circ S_{n2 \rightarrow 3}]) \]
\[ S_u = S_{u2} \circ S_{u1} \quad S_1 = S_u \circ S_{n1 \rightarrow 3} \quad S_2 = S_u \circ S_{n2 \rightarrow 3} \]
\[ \Phi = \Phi_{u2} \quad \omega = \omega'[S_1] \quad \Sigma = (\Sigma'_1[S_1], \Sigma'_2[S_2]) \]
\[ \Delta; \Gamma; \Phi_1, \Phi_2; \Sigma_1, \Sigma_2 \vdash f \Leftarrow e : \omega + \Phi; \Sigma \]

T-SC-App

Figure 4.5: Typing rule for strategy combinator application.

\[ [a] \tau \cdot e_1 \text{ works as a Boolean predicate which potentially triggers runtime failure of this rewrite rule. If } e_1 \text{ evaluates to } (\text{True \{\}}), \text{ the value of } e_2 \text{ packed in a success case will be returned, otherwise a failure case hidden by the rule return type will be returned. To allow the usage of rule parameters in } e_1, \text{ they are put into the typing environment with their traces erased.} \]

**Typing Rules for Strategy Combinators**

The typing rules for strategy combinators are defined following a similar mechanism as the typing rules for rewrite rules, except that we are now focusing on the strategy combinator environment. The most important, also the most complex rule is T-SC-App, which is shown in Figure 4.5. Other typing rules are listed in Figure 4.20 in Section 4.A.3 instead. For the T-SC-App rule, \( f \) is a strategy combinator accepting a strategy with parent bunch \( \phi_1 \), and \( e \) is a strategy with parent bunch \( \phi_2 \). Knowing that \( \Sigma'_2 \) is linear and the strategy environment is empty, it is safe to assert the parent bunch of \( \Phi'_2 \) is \( \phi_2 \) because all resources in \( \Sigma'_2 \) must be used to get \( e \), whose type has \( \phi_2 \) as its parent bunch, and there is nothing else to be traced by \( \Phi'_2 \). By multiplying \( \phi_1 \) and \( \phi_2 \), we enumerate all possibles ways to connect \( e \) and the parameter of \( f \), represented by \( \phi_3 \), and then the corresponding types are unified to rule out invalid traces caused by mismatched patterns.
Values \( v := l v \ | \ \{ (l_i : v_i)_{i \in N} \} \ | \ \text{lam} \ x \to e \)

Eval. Contexts \( E := [ ] \ | \ E e \ | \ \text{let} \ f = E \ \text{in} \ e \ | \ l E \ | E.[] \ | \ (l_i : v_i)_{i \in N} \ | \ l.[] \ | \ E.\cdot \ | \ E.\cdot - l \ | \ \text{match} \ E \ \text{with} \ \langle [] \to e \rangle \ | \ \text{match} \ E \ \text{with} \ \langle l x_1 \to e_1 \ | x_2 \to e_2 \rangle \)

Figure 4.6: Values and evaluation contexts in Rewrite \( S_t \).

**Typing Rule for run**

The typing rule \( \text{T-Run} \) for the keyword \( \text{run} \) is as follows.

\[
\frac{\Delta; \Gamma \vdash e : \{ \cdot \phi \} \phi \to \tau + \Phi \quad \phi \neq [] \quad \tau' = \text{erase}(\tau)}{\Delta; \Gamma \vdash \text{run} \ e : \langle \text{Success} : \tau' \ | \ \text{Failure} : \{ \cdot \} \ | \cdot \rangle}
\]

This rule works as a bridge connecting the traced and untraced worlds. For an arbitrary strategy \( s \), a constant input \( c \) can be supplied to \( s \) using \( (seq \ a \ s) \), where \( a \) is a strategy accepting an empty record/unit value and returning \( c \). The composition \( (seq \ a \ s) \) (if type checks) will have the required type for \( e \) in \( \text{T-Run} \), that is, \( e \) should be a fully applied strategy. \( \text{run} \ e \) gives an untraced variant type with two cases: the Success case containing the result of \( e \) is returned if the execution succeeds, otherwise we get the Failure case. The precondition \( \phi \neq [] \) makes \( \text{T-Run} \) rejects \( e \) whose parent bunch is empty because in this case \( \text{run} \ e \) will absolutely evaluate to the Failure case and it is safe to reject such useless strategy statically. This precondition is justified by Proposition 1 in Section 4.5.

### 4.4.4 Operational Semantics

The operational semantics of Rewrite \( S_t \) is given as a combination of a translation operation in Figure 4.21 in Section 4.A.3, values and evaluation contexts in Figure 4.6 and a reduction relation inherited from Chapter 3 (see Figure 3.12). The translation \( (\{ \cdot \}) \) converts Rewrite \( S_t \) terms (and built-in strategy combinators) to their untraced definitions in the untraced part of Rewrite \( S_t \), which is basically Typed ELEVATE. An exemplar translation rule is \( \{ \text{rule} \ l x \to e \} = (\text{lam} \ x' \to \text{match} \ x' \ \text{with} \ (l x \to e \ | \ r \to \text{Failure} \ \{ \cdot \}) \) translating rewrite rule with variant pattern to ordinary pattern matching, where \( x' \) and \( r \) are fresh. The translation is also defined on types, simply erasing the traces with the erase function. The translation is applied to a whole
Rewrite $S_i$ term/type by performing bottom-up traversal of the AST and applying the translation rule to each sub-AST encountered. The notions of reduction ($\Rightarrow$) and the stepping relation ($\Rightarrow^*$) defined for the small-step operational semantics of Rewrite $S_i$ are directly inherited from Chapter 3.

### 4.5 Technical Results

#### 4.5.1 Type Soundness

We present the type soundness result for our row-polymorphic tracing type system as a corollary of subject reduction and progress theorems.

**Theorem 4.5.1** (Subject Reduction). If $\Delta \vdash e_1 : t \Phi$, and $\Delta' = [\Delta], e'_1 = [e_1], t' = [t], \Delta' + e'_1 : t'$, and $e'_1 \Rightarrow e_2$, then $\Delta' \vdash e_2 : t'$.

*Proof.* The proof proceeds by case analysis on $e'_1 \Rightarrow e_2$. □

**Theorem 4.5.2** (Progress). If $\Delta \vdash e_1 : t \Phi$, and $\Delta' = [\Delta], e'_1 = [e_1], t' = [t], \Delta' + e'_1 : t'$, then either $e'_1$ is a value, or there exists an $\hat{e}$ such that $e'_1 \Rightarrow \hat{e}$.

*Proof.* The proof proceeds by induction on the derivation of $\Delta' + e'_1 : t'$. □

**Corollary 4.5.2.1** (Type Soundness). If $\Delta \vdash e : t \Phi$, and $\Delta' = [\Delta], e' = [e], t' = [t], \Delta' + e' : t'$, then either $e'$ is a value, or there exists an $\hat{e}$ such that $e' \Rightarrow \hat{e}$ and $\Delta' + \hat{e} : t'$.

*Proof.* This is a direct consequence of Theorem 4.5.1 and Theorem 4.5.2. □

#### 4.5.2 On (Un)productive Strategies

**Definition 4.5.1** ((Un)productive strategies). Let $e$ be a well typed strategy with unit input type. If $\Delta \vdash e : \{\cdot//\} \Rightarrow \tau$, then $e$ is unproductive, otherwise $e$ is productive.

If a strategy $e$ is unproductive, the type system with traces will reject its execution $\text{run } e$, due to a direct violation of the premise of $T$-$\text{Run}$. This rejection is sound as an unproductive strategy will always lead to failures when $\text{run}$.

**Proposition 1** (No Unproductive Strategies). If $e$ is unproductive, then $[\text{run } e] \Rightarrow (\text{Failure } \{\})$ and $\text{run } e$ cannot be typed by the type system with traces.
Proof. (Sketch) This proposition states that if the parent bunch of a strategy \( e \) is empty, then \( \text{run} \ e \) will always result in a runtime failure of strategy.

There are only two cases of empty bunches: \text{T-R-Lam-Label} in Figure 4.19 and \text{unifyBunch} in Figure 4.17. Below we show how they lead to a runtime failure:

- \text{T-R-Lam-Label} in Figure 4.19. As explained in Section 4.4.3, the \text{clear} function ensures that \( \rho \) from the parameter type \( \langle l_{[\alpha]} : \tau_x | \rho \rangle \) of the strategy \text{rule} \( l \ x = e \) must be completely traced by the empty bunch. According to the translation in Figure 4.21, any input matching \( \rho \) goes into the failure case.

- \text{unifyBunch} in Figure 4.17. As stated in Section 4.4.3, a trace variable unifies with the empty bunch if no unification target, that is, its parent trace variable or another child trace variable with the common parent, can be found. Since trace multiplication enumerates all possible connections, a trace variable should always be able to find its unification target \text{if it exists}. Thus, the absence of unification target indicates the existence of the empty bunch, which can only be created by other \text{unifyBunch} process or the \text{T-R-Lam-Label} rule. In the later case, the empty bunch corresponds to the failure case.

\( \square \)

4.6 Discussion

In this section, we discuss the type system of Rewrite \( S_t \). With the novel idea of tracing, Rewrite \( S_t \) still preserves as many of the useful features of Typed ELEVATE as possible. For describing the shapes of rewritten programs, Rewrite \( S_t \) keeps the row polymorphic variant and record types in Type ELEVATE. In the definition of rewrite rules, Rewrite \( S_t \) inherits the style of constructing complex patterns with nested simple patterns from Typed ELEVATE. In the rewrite condition, Rewrite \( S_t \) provides an untraced type system which is almost (a subset of) Typed ELEVATE. Furthermore, in the operational semantics, Rewrite \( S_t \) terms are translated into Typed ELEVATE terms for execution and the reduction relation of Typed ELEVATE is re-used.

Certainly, some Typed ELEVATE language features such as recursive definitions and recursive types are absent from Rewrite \( S_t \) due to the restrictions caused by the newly introduced tracing mechanism. Besides, Rewrite \( S_t \) itself also has significant shortcomings. In the rest of this section, we highlight these shortcomings and restrictions and explain where they come from. Specifically, we highlight three main
weakness: weak formal guarantees, a complex formalization, and limited flexibility. We will aim to address these shortcomings in the next chapter.

4.6.1 Weak Formal Guarantees

As shown in Section 4.4, beside type soundness, we make a formal statement showing that unproductive strategies will be rejected by Rewrite $S_t$. This property can help statically detect strategy compositions which will definitely fail at runtime. However, this is a *negative* description about the programs rejected by Rewrite $S_t$, and it is usually more interesting to see the *positive* description about the properties of type-checked programs. Unfortunately, for Rewrite $S_t$, we are not able to do so because a strategy typed with non-empty traces can still fail due to different reasons.

- The most direct reason of failure is the *False* rewrite condition. The rewrite condition can be considered as an untraced sub-expression whose value decides if the rewrite result can be successfully returned. In other words, the rewrite condition affects the rewrite result without affecting the traces. For example, \texttt{rule \{\} $\rightarrow$ return (False \{\} \{\}} always fails because of the constant *False* rewrite condition even though it simply rewrites an unit to an unit and is typed with non-empty traces ($\{\cdot_{\lbrack\alpha_1\rbrack}\} \xrightarrow{\lbrack\alpha\rbrack} \{\cdot_{\lbrack\alpha_2\rbrack}\}$). It is worth noting that the rewrite condition does not need to be a constant *False* \{\}. Actually, any rewrite condition evaluating to *False* \{\} makes the rule fail.

- The behaviours of strategy combinators can also make non-empty traced strategies fail. For example, we can consider the composition $\text{seq} \Leftarrow (lChoice \Leftarrow s_1 \Leftarrow s_2) \Leftarrow s_3$, where $s_1$, $s_2$ and $s_3$ are all non-empty traced strategies and the whole composition is also non-empty traced. Thus, according to the typing rule for the strategy combinator application, we know that at least one of $\text{seq} \Leftarrow s_1 \Leftarrow s_3$ and $\text{seq} \Leftarrow s_2 \Leftarrow s_3$ is non-empty traced because otherwise the whole composition will be empty-traced. For this example, we assume that $\text{seq} \Leftarrow s_1 \Leftarrow s_3$ is empty-traced (so it must fail at runtime), while $\text{seq} \Leftarrow s_2 \Leftarrow s_3$ is not and it can successfully return a rewrite result when a suitable input is supplied. Thus, to make the whole composition with $lChoice$ succeed, we need an input on which $s_1$ fails but $s_2$ succeeds: $lChoice \Leftarrow s_1 \Leftarrow s_2$ will try $s_1$ on that input and fail, and then apply $s_2$ to the input, and finally pass the output to $s_3$. However, such input may never exist. As long as all the successful inputs for $s_2$ are also successful inputs for $s_1$, it will be impossible to construct the input we need here.
In this case, this composition always fails for arbitrary inputs. Section 5.4.2 will discuss this topic in more detail and with a concrete example.

Besides, another cause of the weak formal guarantees provided by Rewrite $S_t$ is its translation-based operational semantics. In Rewrite $S_t$, the traced source terms are translated to untraced target terms which is a subset of the language, and then the operational semantics are defined only for the target terms. Compared with operational semantics defined directly on the source terms, the translation removes traces and strategy-specific constructs, so the related formal properties such as subject reduction are mainly about untraced terms. Specifically, in the subject reduction theorem, both the stepping and the preservation of types happen on the target terms. This is a weakness of Rewrite $S_t$ because traces are excluded from type soundness while they are important language components.

### 4.6.2 Complex Formalization

The complete formalization of Rewrite $S_t$ is lengthy and obviously very complex. Such complex formalization is not only hard to follow, but also unfriendly to amendments and extensions. Listed below are three major reasons for the complexity.

- To support the rewrite condition and the translation-based operational semantics, an untraced row-polymorphic type system, which is roughly Typed EL-EVATE in Chapter 3, is included in the formalization. Although some of the kinding rules are shared by the traced and untraced parts, the inclusion of a standalone type system certainly increase the overall complexity. On the other hand, the rewrite condition is not a key component of the tracing mechanism, and for a core calculus highlighting traces, such feature could or should be removed. Thus, the support of the rewrite condition is unnecessary.

- To uniformly trace variants and records, the design choice of tracing rows in Rewrite $S_t$ creates numerous trace variables. For all types that can be traced (types of kind $\mathcal{T}_v$, more specifically, variant and record types), they are constructed from rows, which are unordered collections of label and type pairs. If we regard a row as a map (data structure), the label-type pairs can be regarded as the key-value pairs. Thus, when we want to use traces to mark different sources contributing to a row, labels and the terminal row variable or empty row (as the base cases of inductively constructing rows) immediately become
suitable positions to locate the trace variables. Consequently, when the size of type grows, the number of trace variables increases with it. An example of this phenomenon can be observed in the typing rule $T\text{-R-Lam-Record}$ where $\text{distr}$ is used to distribute fresh trace variables over types, and the number of trace variables increases linearly with the size of the type. Besides the consumption of print space and ink, managing these trace variables indeed make it subtle to describe the formalization.

- Many trace-related operations in Rewrite $S_t$ are based on substitutions. A representative example is the $T\text{-SC-App}$ rule, where trace multiplication, $\text{norm}$ and $\text{unify}$ are used, and all these three operations provide substitutions in their results. While there are already multiple substitutions, they are further composed to create even more substitutions, and we need to appropriately apply them to get the final type. This way of expressing trace computations is not only hard to read, but also error-prone. Besides, the instantiation of type schemes is another place where substitutions are mainly used. During the instantiation process, potentially traced type or row variables are substituted with untraced types or rows, and fresh trace variables are generated as needed to maintain the tracing structure. To ensure the consistency of traces during type scheme instantiations, type boards are introduced as a cache to store intermediate traced types. However, this new concept of type board is ad hoc, and it also increases the complexity of the formalization.

There are also other design choices or features increasing the complexity of the formalization. For example, $T\text{-R-Lam-Record}$ is a complicated typing rule, but the record patterns are actually not commonly used while matching ASTs and in most cases, simpler tuple pattern would be sufficient.

### 4.6.3 Limited Flexibility

There are two major restrictions in Rewrite $S_t$: the linearity of the strategy combinator variables and the lack of recursive types. These two restrictions are related because non-linear use of strategy combinator variables can easily lead to recursive types, but neither of them is the direct cause of the other. Even for Core $S_t$ in Chapter 5, which provides the support of recursive types, the linear restriction still exists. It is a persistent limitation because it is a trade-off for precise tracing. A more detailed discussion about the linear restriction is deferred until Section 5.3.2.
For the lack of recursive types, as mentioned in Section 4.4, occurrence checks are performed to prevent recursive types because the way of encoding traces in Rewrite $S_t$ cannot properly handle the folding / unfolding of recursive types. For example, we can consider the simple recursive variant type $v$ as $\langle A : v \mid \alpha \rho \rangle$, which will be $\langle A : v$ as $\langle A : v \mid \alpha \rho \rangle \mid \alpha \rho \rangle$ if unfolded once. Formally, without traces, the folding / unfolding of the recursive variant types can be described by the following equivalence.

$$v \text{ as } \langle \rho \rangle \simeq \langle \rho \rangle [v \mapsto v \text{ as } \langle \rho \rangle]$$

Here we inherit the usage of equi-recursive types from Chapter 3. $v$ as $\langle A : v \mid \alpha \rho \rangle$ is the solution for the equation $v^z = \langle A : v^z \mid \alpha \rho \rangle$, which can easily occur during the typing process for compositions such as the following one.

\[
\begin{align*}
\text{lChoice} & \Leftarrow (\text{rule } x \rightarrow \text{return (True \ {\{} \})} \ x) \\
& \Leftarrow (\text{rule } z \rightarrow \text{return (True \ {\{} \})} \ (A \ z))
\end{align*}
\]

In this lChoice composition, the first operand (named as rule A) is the identity rule which will require its LHS to have the same type as its RHS, so when it is composed with the second operand (named as rule B), the LHS type ($v^z$) and the RHS type ($\langle A : v^z \mid \alpha \rho \rangle$) of rule B are unified, giving rise to the equation $v^z = \langle A : v^z \mid \alpha \rho \rangle$ and the recursive solution $v$ as $\langle A : v \mid \alpha \rho \rangle$.

It is worth noting that we are actually unable to trace this composition in Rewrite $S_t$ because of the recursive solution, so the typing process above intentionally ignored traces. By substituting $v^z$ with $v$ as $\langle A : v \mid \alpha \rho \rangle$ in the equation

$$v^z = \langle A : v^z \mid \alpha \rho \rangle$$

we get

$$v \text{ as } \langle A : v \mid \alpha \rho \rangle = \langle A : v \text{ as } \langle A : v \mid \alpha \rho \rangle \mid \alpha \rho \rangle$$

By applying unfolding on the LHS, we get

$$\langle A : v \text{ as } \langle A : v \mid \alpha \rho \rangle \mid \alpha \rho \rangle \equiv \langle A : v \text{ as } \langle A : v \mid \alpha \rho \rangle \mid \alpha \rho \rangle$$

whose both sides are syntactically equivalent (as expressed by the symbol $\equiv$), verifying that $v$ as $\langle A : v \mid \alpha \rho \rangle$ is truly a solution to Equation (4.1). However, this is not the case if we introduce traces.

When we want to trace $v$ as $\langle A : v \mid \alpha \rho \rangle$, the first problem we will face is how to trace the special type variable $v$ representing the type itself. Syntactically, $v$ is a type variable bound by the keyword as, so we may want to trace it with a child
trace variable $\alpha_0$ belonging to the parent trace variable $\alpha$. Since $\upsilon$ may have multiple occurrences in the type, we choose to trace it at its binding site for consistency. Thus, we have $v_{[\alpha_0]}$ as $\langle A : \upsilon \mid \alpha^0 \rangle$ which will be the type for the variable $z$ in rule B. Subsequently, according to T-R-Label, T-R-Return, and T-R-Lam-Var, we can assign the type $v_{[\alpha_0]}$ as $\langle A : \upsilon \mid \alpha^0 \rangle$ to rule B.

To complete the typing of the whole composition, we still need to show that we can apply the unfolding from Equation (4.2) to Equation (4.3) in the presence of traces. In other words, we need to make the underlying type of both sides of rule B syntactically identical without breaking the typing process. A straightforward attempt would be applying the unfolding like a type instantiation described by T-Inst-Type. This would preserve the typing of rule B, but both occurrences of $v_{[\alpha_0]}$ as $\langle A : \upsilon \mid \alpha^0 \rangle$ will be unfolded and we lose the equivalence of underlying types as shown below.

\[
\text{erase}(\langle A_{[\alpha_1]} : v_{[\alpha_2]} \text{ as } A : \upsilon \mid \alpha^0 \rangle \mid \alpha_{[\alpha_1]}^0) \neq \ \\
\text{erase}(\langle A_{[\alpha]} : A_{[\alpha_1]} : v_{[\alpha_2]} \text{ as } A : \upsilon \mid \alpha^0 \rangle \mid \alpha_{[\alpha_1]}^0 \mid \alpha_{[\alpha]}^0) \tag{4.4}
\]

On the other hand, if we only apply unfolding on one occurrence of $v_{[\alpha_0]}$ as $\langle A : \upsilon \mid \alpha^0 \rangle$, we can get equivalent underlying types, but the typing process is broken because the variable $z$ has different types at its binding site ($\langle A_{[\alpha_1]} : v_{[\alpha_2]} \text{ as } A : \upsilon \mid \alpha^0 \rangle \mid \alpha_{[\alpha_1]}^0$) and use site ($v_{[\alpha_0]}$ as $\langle A : \upsilon \mid \alpha^0 \rangle$). Generally, with the current design of tracing mechanism in Rewrite $S_t$, we can only choose either to consistently unfold or to maintain the equations arising from the unification of underlying types. As a result, we cannot trace compositions requiring recursive types like the example, even if they are very simple.

### 4.7 Conclusion

In this chapter, we present Rewrite $S_t$, a type system with a novel tracing mechanism. Starting with several examples of unproductive strategies which cannot be detected in Typed ELEVATE, we presented a case study demonstrating how traces in Rewrite $S_t$ is used to detect unproductive compositions of strategies. We discussed the formal definition and properties of Rewrite $S_t$ and showed that unproductive strategies are rejected by our type system with traces because they are guaranteed to lead to failures when executed. We also discussed the limitations of Rewrite $S_t$, such as weak formal guarantees, complex formalization and limited flexibility. In the next chapter, an improved language, Core $S_t$, will be presented, addressing most of the issues.
Appendix 4.A  Formalization

4.A.1  Typing Rules for Untraced Terms

\[
\begin{align*}
\Gamma, x : \sigma & \vdash t \vdash \Phi \\
\Delta; \Gamma & \vdash x : t + \Phi & \text{T-Var} \\
\Delta; \Gamma & \vdash l e : \tau \mid \rho & \text{T-Label} \\
\Delta; \Gamma & \vdash f \ : t_1 \rightarrow t_2 & \text{T-App} \\
\Delta; \Gamma, x : t & \vdash e \vdash \Phi & \text{T-App} \\
\Delta; \Gamma, l e & \vdash \langle l \ : \tau \mid \rho \rangle & \text{T-Label} \\
\Delta; \Gamma & \vdash f e \ : t_2 & \text{T-App} \\
\Delta; \Gamma & \vdash \text{let } f = e_1 \text{ in } e_2 : t_2 & \text{T-Let} \\
\Delta; \Gamma & \vdash \text{let } f = e_1 \text{ in } e_2 : t_2 & \text{T-Let-Trace} \\
\end{align*}
\]

Figure 4.7: Typing rules for basic untraced terms.

Figure 4.7 shows for the \( \lambda \)-calculus subset of Rewrite \( S_t \) (variables, term applications, lambda abstractions), it also includes the typing rules for ordinary and traces-capturing let-bindings (\text{T-Let} and \text{T-Let-Trace}), and variant constructions (\text{T-Label}). All of them are straightforward. The internal detail of the type scheme instantiation relation \( \leq \) is introduced in Section 4.A.2. Besides, the typing rules for record operations and untraced pattern matchings are basically inherited from Chapter 3 (see Section 3.3.2), so they are not included in this chapter.

4.A.2  Type Scheme Instantiation and Type Boards

Before defining the type scheme instantiation relation, we need a special construct, type board, which allows type instantiation to work in the existence of traces.

\[
\begin{align*}
\text{Type Boards} & \quad \hat{t} := t \mid \rho \\
& \quad \tilde{\tau} := \tau \mid \rho \\
& \quad \beta := \hat{t} \mid (\alpha, \tilde{\tau}), \beta
\end{align*}
\]

Figure 4.8: Syntax of type boards
Figure 4.8 shows the syntax of type boards. Type boards ($\beta$) are only used internally and they will not appear in types or schemes. Starting from an untraced base type, a type board records different ways (indexed by trace variables) of adding traces to the base type. More detailed usage of type boards are explained below.

The type scheme instantiation relation $\leq$ used by T-Var is defined in Figure 4.9. The judgement $\Delta; \Phi \vdash \sigma \leq \sigma' + \Phi'$ means the type scheme $\sigma$ is instantiated to $\sigma'$ starting from the kinding environment $\Delta$ and the tracing environment $\Phi$, and this instantiation produces a new tracing environment $\Phi'$. At the start of instantiation, we substitute all trace variables captured in the tr keyword with freshly generated ones, and then perform instantiation of the type variables captured by $\forall$ with type boards. The definition of subst and related functions can be found in Figure 4.10 and Figure 4.11. Informally, a type board remembers the association between trace variables and traced types to make sure the substitution is consistent. For example, starting from a type variable $a$, it may be traced as $a_{[x_0,y_0]}$ and $a_{[x_0,z_0]}$ at different locations in a larger type. During the execution of subst($\sigma$, $a$, $\beta$, $\Phi$), which means in $\sigma$ substituting $a$ with a type synthesized from type board $\beta$ with the tracing environment $\Phi$, if $a_{[x_0,y_0]}$ is encountered, $\beta$ will be queried for $a_{[x_0]}$ firstly. If $a_{[x_0]}$ does not exist in $\beta$, a new type will be generated using distr and added to $\beta$. Then the same process is performed for $a_{[y_0]}$, and the type replacing $a_{[x_0,y_0]}$ will be synthesized based on the
results. Subsequently, when \( a_{[x_0, z_0]} \) is encountered, previous result from \( a_{[x_0]} \) stored in \( \beta \) can be directly used, so the substitution for \( a_{[x_0]} \) is consistent throughout the instantiation.

Aside from tracing, as shown by \text{T-Inst-Row}, a universally quantified row variable can only be instantiated by a row whose kind is the subset of the variable kind, equivalently speaking, by a “not more general” row. This rule ensures that well-formed rows in a type scheme are still well-formed after instantiation. Besides, the instantiation rules also perform occurrence check to prevent recursive types (which are not supported yet in Rewrite \( S_t \)).

\[
\begin{align*}
\rho_1 + \rho_2 &= \rho \quad \rho_1 + \rho_2 = \rho \\
\{ \rho \} &= \{ \rho_1 \} + \{ \rho_2 \} \\
\langle \rho \rangle &= \langle \rho_1 \rangle + \langle \rho_2 \rangle \\
\alpha_{[\phi_1, \phi_2]} &= \alpha_{\phi_1} + \alpha_{\phi_2} \\
\cdot [\phi_1, \phi_2] &= \cdot \phi_1 + \cdot \phi_2 \\
\phi_1 + \phi_2 &= \rho_1 + \rho_2 \\
\tau_1 + \tau_2 &= \tau \\
\langle l_{[\phi_1, \phi_2] : \tau} | \rho \rangle &= \langle l_{\phi_1 : \tau_1} | \rho_1 \rangle + \langle l_{\phi_2 : \tau_2} | \rho_2 \rangle
\end{align*}
\]

Figure 4.10: Synthesis and query of type boards.
\[
\begin{align*}
(\tilde{t}, \beta', \Phi') &= \text{lookup}(\phi, \beta, \Phi) & (\tilde{t}, \beta', \Phi') &= \text{lookup}(\square, \beta, \Phi) \\
(\tilde{t}, \beta', \Phi') &= \text{subst}(\alpha, \alpha, \beta, \Phi) & (\tilde{t}, \beta', \Phi') &= \text{subst}(\alpha, \alpha, \beta, \Phi) \\
(\alpha, \beta, \Phi) &= \text{subst}(\alpha, \alpha, \beta, \Phi) & (\alpha, \beta, \Phi) &= \text{subst}(\alpha, \alpha, \beta, \Phi) \\
(\rho, \beta', \Phi') &= \text{subst}(\rho, \alpha, \beta, \Phi) & (\rho, \beta', \Phi') &= \text{subst}(\rho, \alpha, \beta, \Phi) \\
\{\rho\}, \beta', \Phi' &= \text{subst}\{\rho\}, \alpha, \beta, \Phi) & \langle \rho\rangle, \beta', \Phi' &= \text{subst}\langle \rho\rangle, \alpha, \beta, \Phi) \\
(\tau', \beta', \Phi') &= \text{subst}(\tau, \alpha, \beta, \Phi) & (\rho', \beta'', \Phi'') &= \text{subst}(\rho, \alpha, \beta', \Phi') \\
((\tilde{t}_\phi \vdash \tau' | \rho'), \beta'', \Phi'') &= \text{subst}((\tilde{t}_\phi \vdash \tau | \rho), \alpha, \beta, \Phi) \\
(t'_1, \beta', \Phi') &= \text{subst}(t_1, \alpha, \beta, \Phi) & (t'_2, \beta'', \Phi'') &= \text{subst}(t_2, \alpha, \beta', \Phi') \\
((t'_1 \rightarrow t'_2), \beta'', \Phi'') &= \text{subst}((t_1 \rightarrow t_2), \alpha, \beta, \Phi) \\
(\tau'_1, \beta', \Phi') &= \text{subst}(\tau_1, \alpha, \beta, \Phi) & (\tau'_2, \beta'', \Phi'') &= \text{subst}(\tau_2, \alpha, \beta', \Phi') \\
((\tau'_1 \rightarrow \tau'_2), \beta'', \Phi'') &= \text{subst}((\tau_1 \rightarrow \tau_2), \alpha, \beta, \Phi) \\
(\tau'_1, \beta', \Phi') &= \text{subst}(\tau_1, \alpha, \beta, \Phi) & (\tau'_2, \beta'', \Phi'') &= \text{subst}(\tau_2, \alpha, \beta', \Phi') \\
(\omega', \beta'', \Phi'') &= \text{subst}(\omega, \alpha, \beta'', \Phi'') \\
(((\tau'_{\phi_1} \rightarrow \tau'_{\phi_2}) \Rightarrow \omega'), \beta'', \Phi'') &= \text{subst}(((\tau_1 \rightarrow \tau_2) \Rightarrow \omega), \alpha, \beta, \Phi) \\
(\forall \ (\alpha : \kappa). \ \beta, \Phi) &= \text{subst}(\forall \ (\alpha : \kappa). \ \beta, \Phi) \\
(\overline{\alpha}, \beta', \Phi') &= \text{subst}(\overline{\alpha}, \alpha, \beta, \Phi) & (\overline{\alpha}, \beta', \Phi') &= \text{subst}(\forall \ (\alpha' : \kappa). \ \beta, \Phi)
\end{align*}
\]

Figure 4.11: Substitution with type boards.
### Other Definitions and Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi + [\alpha, \phi_1] = \phi_2$</td>
<td>$\alpha \in \phi$ --------- $\phi + \phi_1 = \phi_2$</td>
</tr>
<tr>
<td>$\phi + [] = \phi$</td>
<td>$\alpha \notin \phi$ --------- $\phi + [\alpha, \phi_1] = [\alpha, \phi_2]$</td>
</tr>
</tbody>
</table>

- $[] = \text{parent}([], \Phi)$
- $[\alpha, \phi_2] = \text{parent}([\alpha, \phi_1], \phi \triangleright \Phi)$
- $(\alpha : \alpha') \in \Phi$ $\phi_2 = \text{parent}(\phi_1, \phi \triangleright \Phi)$
- $[\alpha', \phi_2] = \text{parent}([\alpha, \phi_1], \phi \triangleright \Phi)$

Figure 4.12: Auxiliary definitions.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi \triangleright \phi$</td>
<td>$\cdot \triangleright []$</td>
</tr>
<tr>
<td>$\Phi \triangleright ([\alpha'] + \phi)$</td>
<td>$(\alpha : \alpha', \Phi) \triangleright ([\alpha'] + \phi)</td>
</tr>
<tr>
<td>$\Phi \triangleright (\phi' + \phi)$</td>
<td>$(\alpha : \phi', \Phi) \triangleright (\phi' + \phi)$</td>
</tr>
<tr>
<td>$\Phi \triangleright \phi$</td>
<td>$(\phi \triangleright \Phi) \text{ OK}$</td>
</tr>
</tbody>
</table>

Figure 4.13: Well-formedness rules for tracing environments.
Figure 4.14: Erasure and clearance of traces.
Chapter 4. Traces for Safe Compositions

Figure 4.15: Kinding rules.
<table>
<thead>
<tr>
<th>fresh $\alpha'$</th>
<th>$(\phi, \rho') = \text{distr}(\rho)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$([\alpha'], \alpha_{[\alpha']}) = \text{distr}(\alpha)$</td>
<td>$(\phi, {\rho'}) = \text{distr}{\rho}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$(\phi, \rho') = \text{distr}(\rho)$</th>
<th>fresh $\alpha'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\phi, \langle \rho' \rangle) = \text{distr}(\langle \rho \rangle)$</td>
<td>$([\alpha'], \langle \alpha' \rangle) = \text{distr}\langle \cdot \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fresh $\alpha'$</th>
<th>$(\phi_{\rho}, \rho') = \text{distr}(\rho)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\phi_{\tau}, \tau') = \text{distr}(\tau)$</td>
<td>$(\alpha', \phi_{\rho}, \phi_{\tau}, (l_{[\alpha']} : \tau'</td>
</tr>
</tbody>
</table>

Figure 4.16: Distribution of traces
\[ S = \text{unifyBunch}(\Phi, \phi, [(\alpha_j)_{j \in M}]) \]
\[
\forall j \in M, S_j = \text{unifyTrace}(\Phi, \alpha, \alpha_j)
\]
\[
() = (S_j \circ)_{j \in M}()
\]
\[
(\alpha \mapsto []) \circ S = \text{unifyBunch}(\Phi, [\alpha, \phi, [(\alpha_j)_{j \in M}])
\]

\[ S = \text{unifyBunch}(\Phi, \phi, [(\alpha_j)_{j \in M}]) \]
\[ \forall j \in M, S_j = \text{unifyTrace}(\Phi, \alpha, \alpha_j) \]
\[ S' = (S_j \circ)_{j \in M}() \quad S' \neq () \]
\[ S' \circ S = \text{unifyBunch}(\Phi, [\alpha, \phi, [(\alpha_j)_{j \in M}]) \]
\[ S = \text{unifyBunch}(\Phi, [], [(\alpha_j)_{j \in M}]) \]

\[ S = \text{unifyBunch}(\Phi, \phi_1, \phi_2) \]
\[ (\Phi', S') = \text{norm}(\Phi[S]) \]
\[ (\Phi', S' \circ S) = \text{unify}(\Phi, \alpha_{\phi_1}, \alpha_{\phi_2}) \]
\[ (\Phi', S) = \text{unify}(\Phi, \rho_1, \rho_2) \]
\[ (\Phi, S) = \text{unify}(\Phi, \{\rho_1\}, \{\rho_2\}) \]

\[ S_1 = \text{unifyBunch}(\Phi, \phi_1, \phi_2) \]
\[ (\Phi_1, S'_1) = \text{norm}(\Phi[S_1]) \]
\[ (\Phi_2, S_2) = \text{unify}(\Phi_1, \tau_1 [S'_1 \circ S_1], \tau_2 [S'_1 \circ S_1]) \]
\[ (\Phi_3, S_3) = \text{unify}(\Phi_2, \rho_1 [S_2 \circ S'_1 \circ S_1], \rho_2 [S_2 \circ S'_1 \circ S_1]) \]
\[ (\Phi_3, S_3 \circ S_2 \circ S'_1 \circ S_1) = \text{unify}(\Phi, (l_{\phi_1} : \tau_1 | \rho_1), (l_{\phi_2} : \tau_2 | \rho_2)) \]

Figure 4.17: Unification of traced types.
\[
(\phi \triangleright \Phi, S) = \text{norm}(\phi \triangleright \Phi)
\]

\[
((\phi \triangleright \alpha : \alpha', \Phi'), S) = \text{norm}(\phi \triangleright \alpha : \alpha', \Phi)
\]

\[
(\phi \triangleright \Phi', S) = \text{norm}(\phi \triangleright \Phi) \quad \text{fresh} \quad (\alpha_k)_{k \in P} \quad S' = \alpha \mapsto [(\alpha_k)_{k \in P}]
\]

\[
((\phi \triangleright (\alpha_k : \alpha'_k)_{k \in P}, \Phi'), S' \circ S) = \text{norm}(\phi \triangleright \alpha : [(\alpha'_k)_{k \in P}], \Phi)
\]

Figure 4.18: Normalization rules for tracing environments.
If $\Phi = [] \vdash \cdot$ Then fresh $\alpha$ $\overline{\Phi} = \cdot$. Else $\Phi = [\alpha] \triangleright \overline{\Phi}$

\[
(\{\alpha_i\}_{i \in N}, \tau_{new}) = \text{distr} (\text{erase} (\tau_x)) \\
\Phi_{ext} = [\alpha] \triangleright \overline{\Phi}, (\alpha_i : \alpha)_{i \in N}
\]

\[
\Delta; \Gamma; \Phi_{ext}; \Theta, x : \tau_{new}^{[\alpha]} \vdash e : [\alpha] \tau_e \rightarrow \Phi'; \Theta', x : \tau_x^{[\alpha]}
\]

\[
\Delta; \Gamma; \Phi; \Theta \vdash \text{rule} \; x \rightarrow e : \tau_x^{[\alpha]} \overset{\tau_e \rightarrow \Phi'}{\rightarrow} \Phi'; \Theta'
\]

T-R-Lam-Var

If $\Phi = [] \vdash \cdot$ Then fresh $\alpha$ $\overline{\Phi} = \cdot$. Else $\Phi = [\alpha] \triangleright \overline{\Phi}$

\[
(\{\alpha_i\}_{i \in N}, \tau_{new}) = \text{distr} (\text{erase} (\tau_x)) \\
\Phi_{ext} = [\alpha] \triangleright \overline{\Phi}, (\alpha_i : \alpha)_{i \in N}
\]

\[
\rho = \text{clear} (\rho) \\
\Delta; \Gamma; \Phi_{ext}; \Theta, x : \tau_{new}^{[\alpha]} \vdash e : [\alpha] \tau_e \rightarrow \Phi'; \Theta', x : \tau_x^{[\alpha]}
\]

\[
\Delta; \Gamma; \Phi; \Theta \vdash \text{rule} \; l \; x \rightarrow e : \langle [l_{\alpha}] : \tau_x \mid \rho \rangle^{[\alpha]} \overset{\tau_e \rightarrow \Phi'}{\rightarrow} \Phi'; \Theta'
\]

T-R-Lam-Label

If $\Phi = [] \vdash \cdot$ Then fresh $\alpha$ $\overline{\Phi} = \cdot$. Else $\Phi = [\alpha] \triangleright \overline{\Phi}$

\[
\text{fresh } \alpha_1 \\
\Phi_{ext} = [\alpha] \triangleright \overline{\Phi}, \alpha_1 : \alpha
\]

\[
\Delta; \Gamma; \Phi_{ext}; \Theta, \star : \alpha_1 \vdash e : [\alpha] \tau_e \rightarrow \Phi'; \Theta', \star : \alpha_1
\]

\[
\Delta; \Gamma; \Phi; \Theta \vdash \text{rule} \; \{ \} \rightarrow e : \{ [\alpha_1] \}^{[\alpha]} \overset{\tau_e \rightarrow \Phi'}{\rightarrow} \Phi'; \Theta'
\]

T-R-Lam-Unit

If $\Phi = [] \vdash \cdot$ Then fresh $\alpha$ $\overline{\Phi} = \cdot$. Else $\Phi = [\alpha] \triangleright \overline{\Phi}$

\[
(\{\alpha_i\}_{i \in N}, \tau_{new}) = \text{distr} (\text{erase} (\tau_x))
\]

\[
(\{\alpha_j\}_{j \in M}, \rho_{new}) = \text{distr} (\text{erase} (\rho))
\]

\[
\Phi_{ext} = [\alpha] \triangleright \overline{\Phi}, (\alpha_i : \alpha)_{i \in N}, (\alpha_j : \alpha)_{j \in M}
\]

\[
\Delta; \Gamma; \Phi_{ext}; \Theta, x : \tau_{new}^{[\alpha]}, r : \{\rho_{new}\}^{[\alpha]} \vdash e : [\alpha] \tau_e \rightarrow
\]

\[
\Phi'; \Theta', x : \tau_x^{[\alpha]}, r : \{\rho\}^{[\alpha]}
\]

\[
\Delta; \Gamma; \Phi; \Theta \vdash \text{rule} \; \{ l : x \mid r \} \rightarrow e : \{ [l_{\alpha}] : \tau_x \mid \rho \}^{[\alpha]} \overset{\tau_e \rightarrow \Phi'}{\rightarrow} \Phi'; \Theta'
\]

T-R-Lam-Record

Cont.
\[
\frac{x : \tau^{[\alpha]} \in \Theta}{\Delta; \Gamma; \Phi; \Theta \vdash x : \tau^{[\alpha]}} \quad \text{T-R-Var}
\]

\[
\frac{\Delta; \Gamma; \Phi; \Theta \vdash f : \tau_1 \xrightarrow{[\alpha]} \tau_2 + \Phi'; \Theta', x : \tau^{[\alpha]}}{\Delta; \Gamma; \Phi; \Theta \vdash x \mapsto f : \tau_1 \xrightarrow{[\alpha]} \tau_2 + \Phi'; \Theta', x : \tau^{[\alpha]}} \quad \text{T-R-App}
\]

\[
\frac{\Phi' = \text{unify}(\Phi', \tau, \tau_1) \quad \tau'_2 = \tau_2 \cdot [S] \quad \tau' = \tau [S]}{\Delta; \Gamma; \Phi; \Theta \vdash x : \tau^{[\alpha]}} \quad \text{T-R-Var}
\]

\[
\frac{\Delta; \Gamma; \Phi; \Theta \vdash \Psi : \tau, e : \tau^{[\alpha]} + \Phi'}{\Delta; \Gamma; \Phi; \Theta \vdash \Psi : \tau, e : \tau^{[\alpha]} + \Phi'} \quad \text{T-R-Label}
\]

\[
\frac{\Theta \neq \cdot \Phi = [\alpha] \triangleright \Phi}{\Delta; \Gamma; \Phi; \Theta \vdash e : \tau^{[\alpha]} + \Phi' \quad \rho = \text{clear}(\rho)} \quad \text{T-R-RecordCons}
\]

\[
\frac{\Theta \neq \cdot \Phi = [\alpha] \triangleright \Phi}{\Delta; \Gamma; \Phi; \Theta \vdash l \cdot e : \langle \lambda l : \tau | \rho \rangle^{[\alpha]} + \Phi'}
\]

\[
\frac{\Phi = [\alpha] \triangleright \Phi \quad \Delta; \Gamma; \Phi; \Theta \vdash e : \tau^{[\alpha]} + \Phi'}{\Delta; \Gamma; \Phi; \Theta \vdash \Phi' = \Phi \quad \Theta' = \Theta}
\]

\[
\frac{\Phi' = [\alpha] \triangleright \Phi, \alpha_1 : \alpha \quad \Theta' = \Theta, \star : \alpha_1 \quad \forall i \in \mathcal{N}, \Delta; \Gamma; \Phi; \Theta' \vdash e_i : \tau_i^{[\alpha]} \quad \Delta; \Gamma; \Phi; \Theta \vdash \{ (l_i : e_i)_{i \in \mathcal{N}} : \{ (l_i^{[\alpha]} : \tau_i)_{i \in \mathcal{N}} | \cdot^{[\alpha]} \} \}^{[\alpha]} + \Phi'}{\Delta; \Gamma; \Phi; \Theta \vdash \text{erase}(\Theta) \vdash l \cdot e : \langle \text{True} : \{ \cdot \} | \text{False} : \{ \cdot \} | \cdot \rangle \quad \text{T-R-Return}}
\]

Figure 4.19: Typing rules for rewrite rules.
\[
\Sigma = x : \tau_1 \to \tau_2 \\
\Delta; \Gamma; \Phi; \Sigma \vdash x : \tau_1 \to \tau_2
\]

\[\Sigma = x : \tau_1 \to \tau_2 \quad \text{T-SC-Var}\]

\[
\Delta; \Gamma; \Phi; \Sigma \vdash f : (\tau_1 \to \tau_2) \Rightarrow \omega' + \Phi_1; \Sigma'
\]

\[
\Delta; \Gamma; \Phi; \Sigma \vdash e : \tau_3 \to \tau_4 + \Phi'_2; \Sigma'_2
\]

\[
(\phi_3, S_{1\to-3}, S_{2\to-3}) = \phi_1 \times \phi_2 \\
(\Phi'_1[S_{1\to-3}]) = \text{norm}(\Phi'_1[S_{1\to-3}])
\]

\[
\phi'_3 \triangleright \Phi'_{n_2}, S_{n_2} = \text{norm}(\Phi'_{n_2}[S_{2\to-3}]) \\
\Phi' = \phi'_3 \triangleright \Phi'_{n_1}, S_{n_2}
\]

\[
S_{n_1\to-3} = S_{n_1} \circ S_{1\to-3} \\
S_{n_2\to-3} = S_{n_2} \circ S_{2\to-3}
\]

\[
(\Phi_{u_1}, S_{u_1}) = \text{unify}(\Phi', \tau_1[S_{n_1\to-3}], \tau_3[S_{n_2\to-3}])
\]

\[
(\Phi_{u_2}, S_{u_2}) = \text{unify}(\Phi_{u_1}, \tau_2[S_{u_1} \circ S_{n_1\to-3}], \tau_4[S_{u_1} \circ S_{n_2\to-3}])
\]

\[
S_u = S_{u_2} \circ S_{u_1} \\
S_1 = S_u \circ S_{n_1\to-3} \\
S_2 = S_u \circ S_{n_2\to-3}
\]

\[
\Phi = \Phi_{u_2} \\
\omega = \omega'[S_1] \\
\Sigma = (\Sigma'[S_1], \Sigma'_2[S_2])
\]

\[
\Delta; \Gamma; \Phi_1, \Phi_2; \Sigma_1, \Sigma_2 \vdash f \iff e : \omega + \Phi; \Sigma
\]

\[\text{T-SC-App}\]

\[
\text{fresh } \alpha \\
\left(\left(\{\alpha_i\} \in \mathbb{N}\right), \tau_{1\text{new}}\right) = \text{distr}(\text{erase}(\tau_1))
\]

\[
\omega_x = \tau_{1\text{new}} \xrightarrow{[\alpha]} \tau_{2\text{new}}
\]

\[
\Phi_{\text{ext}} = \Phi, [\alpha] \triangleright (\alpha_i : \alpha)_{i \in \mathbb{N}}(\alpha_j : \alpha)_{j \in \mathbb{M}}
\]

\[
\Delta; \Gamma; \Phi_{\text{ext}}; \Sigma; x : \omega_x \vdash e : \omega + \Phi'; \Sigma', x : \tau_1 \to \tau_2
\]

\[\text{T-SC-Lam}\]

\[
\Delta; \Gamma; \Phi; \Sigma \vdash \text{st} x \Rightarrow e : (\tau_1 \to \tau_2) \Rightarrow \omega + \Phi'; \Sigma'
\]

Figure 4.20: Typing rules for strategy combinators.
\[[\text{rule } x \rightarrow e] = (\text{lam } x \rightarrow e)\]
\[[\text{rule } l \cdot x \rightarrow e] = (\text{lam } x' \rightarrow \text{match } x' \text{ with } (l \cdot x \rightarrow e \mid r \rightarrow \text{Failure } \{\}))\]
\[[\text{rule } l \cdot x \rightarrow e = (\text{lam } x \rightarrow (\text{lam } r \rightarrow e) \cdot x' \mid l) \cdot x'.l)\]
\[[\text{rule } \{\} \rightarrow e] = (\text{lam } x' \rightarrow \text{match } x' \text{ with } (\{\} \rightarrow e))\]
\[[\text{rule } x \leftarrow e] = (e \cdot x)\]
\[[\text{return } e_1 \cdot e_2] = (\text{match } e_1 \text{ with } (\text{True } \rightarrow \text{Success } e_2 \mid \text{False } \rightarrow \text{Failure } \{\}))\]
\[[\text{st } x \Rightarrow e] = (\text{lam } x \rightarrow e)\]
\[[e_1 \leftarrow e_2] = (e_1 \cdot e_2)\]
\[[\text{run } e] = (e \{\})\]
\[\text{otherwise } [e] = e\]

\[[\text{seq}] = (\text{lam } f \rightarrow \text{lam } s \rightarrow \text{lam } x \rightarrow \text{match } (f \cdot x) \text{ with } \langle \]
\[\text{Success } y \rightarrow s \cdot y \mid \]
\[r \rightarrow \text{match } r \text{ with } (\text{Failure } \rightarrow \text{Failure } \{\}))\]
\[[\text{lChoice}] = (\text{lam } f \rightarrow \text{lam } s \rightarrow \text{lam } x \rightarrow \text{match } (f \cdot x) \text{ with } \langle \]
\[\text{Success } y \rightarrow \text{Success } y \mid \]
\[r \rightarrow \text{match } r \text{ with } (\text{Failure } \rightarrow s \cdot x)\rangle)\]

\[[\tau_1 \rightarrow \tau_2] = \text{erase}(\tau_1) \rightarrow (\text{Success } : \text{erase}(\tau_2) \mid \text{Failure } : \{\} \mid \cdot)\]
\[[t_1 \Rightarrow t_2] = t_1 \rightarrow t_2\]
\[\text{otherwise } [t] = t\]

\[[\mathcal{T}_s] = \mathcal{T} \quad [\mathcal{T}_-] = \mathcal{T} \quad [\mathcal{T}_\exists] = \mathcal{T} \quad [\mathcal{R}] = \mathcal{R}\]

Figure 4.21: Translation.
Chapter 5

Traced Types for Safe Strategic Rewriting

5.1 Introduction

In last chapter, we introduced the idea of using a tracing system to statically estimate the execution of strategies. We formalized the design of a strategy language with a traced type system, Rewrite $S_t$, which as we discussed, has major limitations. In this chapter, we will present Core $S_t$ which also features a traced type system but addresses most of the issues in Rewrite $S_t$ by providing simplified formalization, recursive types, more primitive combinators, and stronger formal guarantees.

In Core $S_t$, the design and presentation of the tracing mechanism is significantly simplified. Instead of tracing labels and type variables, Core $S_t$ directly traces types, and the number of trace variables are reduced without compromising on the precision of tracing. Recursive types are supported in Core $S_t$ because with this new tracing mechanism, it is possible to define the folding / unfolding of equi-recursive types in the presence of traces.

For defining rewrite rules and strategies in Core $S_t$, a new primitive strategy combinator $\textbf{both}$ is provided, allowing the definitions of non-recursive traversal combinators. The operational semantics is now defined directly on the term syntax. Without the need of a separate translation process, the operational semantics of Core $S_t$ is more intuitive.

In Core $S_t$, the formal guarantees provided by the traced type system are stronger. We are able to formally define the meaning of "well-traced" and show that strategies which are structurally well-typed and well-traced are free of composition errors, and
with suitable input, their execution is guaranteed to have a possible successful execution path. However, to support these stronger guarantees, the left-choice combinator which was part of the language primitives is now replaced by the non-deterministic choice combinator. More details about this change will be discussed in Section 5.4.2.

In this chapter, to present our improved traced type system, we show a number of examples demonstrating Core $S_t$ in practice (Section 5.2), and then we present a formalization of Core $S_t$ and its type system (Section 5.3.1–5.3.6) and prove its type soundness (Section 5.3.7–5.3.8). We formally define what it means for well-typed and well-traced strategies “not to go wrong” and we show that well-typed but not well-traced strategies must result in a runtime failure (Section 5.3.9).

It is worth noting that prior type systems for strategy languages, as discussed in Chapter 2, prioritized the support of language features, particularly generic traversals and recursion of strategies, willing to compromise on the precision of the type checking. We make the opposite trade-off: our precise and expressive traced types come at the cost of not (yet) supporting recursive strategies and generic traversals. Section 5.2 shows that our restrictive strategy language is useful in practice and capable of expressing complex examples using ad-hoc traversals. In Section 5.4, we evaluate Core $S_t$ as a strategy language, compare Core $S_t$ with Rewrite $S_t$, discuss our design decisions and trade-offs and explore ways how we might introduce generic traversals and limited forms of recursion in the future. We also discuss other possible extensions for Core $S_t$ in Section 5.5.

5.2 Core $S_t$ by Example

In this section, we present Core $S_t$ by example, starting with discussing valid compositions of rewrites showing the types given to these strategies by our type system. We will then discuss problematic compositions which our type system will flag with warnings in Section 5.2.2 and wrong compositions that our type system rejects with errors in Section 5.2.3.
5.2.1 Valid Strategy Compositions and Strategy Combinators

Example 5.1

We start with a simple composition of three rewrite rules:

\[
\text{let } e_1 = \text{rule } m \times n \rightarrow n \times m ; (\text{rule } 1 \times v \rightarrow v \mid\mid \text{rule } 2 \times w \rightarrow w + w) \]

To define the strategy \( e_1 \), we first swap the two factors \( n \) and \( m \), before either removing an unnecessary multiplication with \( 1 \) or, alternatively, turning a multiplication with \( 2 \) into an addition. Adopting the notations from Chapter 4, we use the sequential combinator (\(;\)) and the choice combinator (\(||\)) here, but as mentioned in Section 5.1, the choice combinator is now non-deterministic, which means it applies one of two strategies in a non-deterministic way, or fails if none is applicable.

To represent the expressions we rewrite, such as \( m \times n \), as types, we encode them using row-polymorphic variant types, as formally explained in Section 5.3. In this section, we will write these structural types informally as we would present them to users of the type system. As with the other type systems in this thesis, rewrite rules and their compositions as strategies are given a strategy type that reflects the syntactic transformation described by the rewrite. For example, the first rewrite rule has the following strategy type:

\[
\text{rule } m \times n \rightarrow n \times m : a_0 \times a_1 \rightarrow a_1 \times a_0
\]

The variables appearing in the type are trace variables. Here all variables belong to a single trace identified by \( a \) indicating that there is only a single possible way to perform the rewrite described by the strategy: swapping the two operands of the multiplication, represented by the two trace variables \( a_0 \) and \( a_1 \).

The strategy type for the choice composition of the two other rewrites is more interesting:

\[
(\text{rule } 1 \times v \rightarrow v \mid\mid \text{rule } 2 \times w \rightarrow w + w) : 1 \times b_0 \mid 2 \times c_0 \rightarrow b_0 \mid c_0 + c_0
\]

Here the strategy type has two traces, \( b \) and \( c \), indicating that there are two possible executions of this rewrite strategy: either, the first rewrite rule transforms \( 1 \times b_0 \) into \( b_0 \), or, the second rewrite rule transforms \( 2 \times c_0 \) into \( c_0 + c_0 \). When there are multiple traces in a strategy type, we write the input and output for each trace individually separated by bars (\( \mid\mid \)) to indicate that these are separated possible executions of the
strategy. In our formal system, types are not represented this way. We present a justification in Section 5.3 why this simplified and intuitive presentation of types is a truthful reflection of the information available in the formal type.

When sequentially combining both strategies, we obtain a strategy type that reflects the effect of the overall strategy:

```plaintext
let e1 = rule m * n -> n * m ;
      (rule 1 * v -> v || rule 2 * w -> w + w) :
      d0 * 1 | e0 * 2 [d,e] d0 | e0 + e0
```

The strategy type now indicates that there are two possible ways to successfully perform the composed strategy: either the input expression has the form $d_0 \times 1$ and it will be rewritten into $d_0$, or, alternatively, the input has the form $e_0 \times 2$ and it will be rewritten into $e_0 + e_0$. For any other expressions as input, the composed strategy will fail. For the whole strategy as a sequential composition, the one trace from the first strategy is fully connected with the two traces from the second strategy, resulting an overall strategy with $1 \times 2 = 2$ traces. Thus, we call this example a fully composed strategy.

Although the concept of tracing here stays similar to Chapter 4, and this example seems straightforward, Rewrite $S_t$ is not able to typecheck this example because type unifications for the choice composition produces recursive types, which are not supported by Rewrite $S_t$, and are hidden by the informal simplified strategy type here. A detailed comparison of the formal and informal strategy types is in Section 5.3.2.

**Example 5.2**

We can also check the application of strategies to the input expressions. Consider the following example where we pass the expression $5 \times 2$ as input to the strategy $e_1$ from Example 5.1 and then pass the resulting expression as input to an additional rewrite rule:

```plaintext
let e2 = (rule 5 + 5 -> 10) (e1 (5 * 2))
```

The application of $5 + 2$ to $e_1$ has the strategy result type: $[b] \triangleright 5+5$. In this case it indicates a successfully rewritten expression, written on the right side of the black triangle. No trace variable $b_i$ appears on the right side, as the expression only contains constants and no variables.

Compared with the untraced strategy execution results in Rewrite $S_t$, the usage
of strategy execution results in Core $S_t$ is more flexible, i.e., the execution result of a strategy can be processed by another strategy, so strategy executions can be cascaded.

With this type as the input for the rule $(\text{rule } 5 + 5 \rightarrow 10) : 5 + 5 \rightarrow 10$, we obtain:

\[
\text{let } e_2 = (\text{rule } 5 + 5 \rightarrow 10) (e_1 (5 * 2)) : [c] \rightarrow 10
\]

This type confirms that this strategy execution is valid and will produce the expression $5 + 5$ which is later rewritten to $10$.

**Example 5.3**

Now we consider a sequential composition of two choice compositions:

\[
\text{let } e_3 = \left(\text{rule } m * n \rightarrow n * m || \text{rule } m + n \rightarrow n + m\right) ; \\
\left(\text{rule } 1 * v \rightarrow v || \text{rule } 0 + w \rightarrow w\right)
\]

Here, the choice composition in the first line swaps the operands of multiplication or addition, the second choice composition in the second line removes the identity element for multiplication or addition. When sequentially composing two choice compositions, in general there are four possible execution paths: left-left, left-right, right-left, and right-right. But clearly, in this example only two will execute successfully: the first one swapping the operands and removing the identity element for multiplication (left-left) and the other for addition (right-right). To reflect this, this example could also have been written as:

\[
\left(\text{rule } m * n \rightarrow n * m ; \text{rule } 1 * v \rightarrow v \right) || \\
\left(\text{rule } m + n \rightarrow n + m ; \text{rule } 0 + w \rightarrow w\right)
\]

Our tracing type system is capable of precisely reflecting all possible execution paths in the type. The individual choice compositions have the types:

\[
\left(\text{rule } m * n \rightarrow n * m \right) : a_0 * a_1 | b_0 + b_1 \rightarrow a_1 * a_0 | b_1 + b_0
\]

and

\[
\left(\text{rule } 1 * v \rightarrow v || \text{rule } 0 + w \rightarrow w\right) : 1 * c_0 | 0 + d_0 \rightarrow c_0 | d_0
\]

When sequentially composing them, our tracing type system connects the traces of both strategy types to compute the possible traces of the composition representing all possible execution paths.
let e3 = (rule m * n -> n * m || rule m + n -> n + m) ;
      (rule 1 * v -> v || rule θ + w -> w) :
      e0 * 1 | f0 + 0 [e,f] e0 | f0

The final type of the composition clearly expresses that the overall strategy rewrites either $e_0 * 1$ into $e_0$ or $f_0 + 0$ into $f_0$. The strategy must fail for all other inputs. For the whole strategy as a sequential composition, the two traces from the first strategy is partially connected with the two traces from the second strategy, resulting an overall strategy with two traces. Thus, we call this example a partially composed strategy.

This example does not produce any recursive types, so Rewrite $S_t$ can also type-check this example and identify the two valid execution paths. Actually, without the operand swapping part, this example is almost the same as the left-choice example in Sections 4.2 and 4.3. For instance, we briefly recall an example from Section 4.3, which can be expressed as follows.

let re1 = rule ? -> θ + z ; (rule 1 * n -> n || rule θ + m -> m)

The output of the first rewrite rule (rule ? -> θ + z) matches the input of one subsequent rewrite rule (rule θ + m -> m), constituting a successful execution path. Thus, the corresponding single-traced type is assigned to this composition.

let re1 = rule ? -> θ + z ; (rule 1 * n -> n || rule θ + m -> m) :
      ? [a] a0

We will discuss in Section 5.3 formally how our type system computes the connected traces of strategy compositions.

Example 5.4

So far we have only seen strategies that are build using the two fundamental combinators ; and ||. But, we can build custom strategy combinators as well, these take strategies as arguments and augment their behaviours. For example, let’s consider the swapOps combinator that takes a strategy $s$ and applies it after swapping the operands of an arbitrary binary operator $op$.

let swapOps = st s => rule m op n -> n op m ; s

This custom combinator can be used to swap the order of operands for any strategy transforming binary expressions, such as: swapOps (rule 1 * v -> v) which has the type: $b_0 * 1 \overset{[b]}{\longrightarrow} b_0$. 
Being similar to Rewrite $S_t$, strategy combinators in Core $S_t$ have a dedicated arrow type $\Rightarrow$. For our custom combinator, its type is:

\[
\text{let} \quad \text{swapOps} = \text{st} \Rightarrow \text{rule} \quad m \circ\!\!\!\!\circ n \Rightarrow n \circ\!\!\!\!\circ m ; \quad s : ((a_0, a_2, a_1) \Rightarrow a_3) \Rightarrow ((a_1, a_2, a_0) \Rightarrow a_3)
\]

This type indicates that the input expression changes its order from $a_0, a_2, a_1$ to $a_1, a_2, a_0$, clearly showing the swap of the operands.

This strategy combinator can be defined in Rewrite $S_t$ similarly.

**Example 5.5**

In this example, we use a third fundamental strategy combinator both ($\&\&$) in addition to $;$ and $||$. The $\&\&$ combinator is for the pairing composition of strategies, that is, $s_1 \&\& s_2$ applies the two argument strategies, $s_1$ and $s_2$, to the elements of a pair respectively. $\&\&$ allows us to construct more complex custom strategy combinators. For example, the code below defines a strategy combinator plusRight which takes a strategy and applies it to the second (or right) operand in a plus expression. More specifically, plusRight is constructed by sequentially composing three rewriting steps: the first step extracts both operands of a plus expression and puts them into a pair; the second step applies the input strategy $s$ to the second element of the pair while keeping the first element unchanged by applying the identity strategy $\text{rule} \ x \Rightarrow x$; the third step takes both elements of the pair and reconstruct the plus expression.

\[
\text{let} \quad \text{plusRight} = \text{st} \Rightarrow \text{rule} \quad m \circ\!\!\!\!\circ n \Rightarrow (m, n) ; \\
\text{rule} \ x \Rightarrow x \&\& s ; \\
\text{rule} \quad (m, n) \Rightarrow m + n : \\
(a_1 \Rightarrow a_2) \Rightarrow (a_0 + a_1 \Rightarrow a_0 + a_2)
\]

The type of plusRight also clearly shows that it takes a strategy of type $a_1 \Rightarrow a_2$ and rewrites $a_0 + a_1$ into $a_0 + a_2$.

Due to the lack of the $\&\&$ combinator, Rewrite $S_t$ cannot express this customized plusRight combinator. Adding the $\&\&$ combinator to Rewrite $S_t$ is realizable, but not very straightforward.
5.2.2 Problematic Strategy Compositions Causing Warnings

After a number of examples showing valid strategies, we will now investigate problematic strategies. In this section, we investigate an example that will still execute correctly, but contains a dead execution branch that our type system detects and warns the user about.

Example 5.6

We consider the following strategy:

```
let e5 = (rule m + n -> n + m ; rule m * n -> n * m) ||
        rule 1 * v -> v
```

This is a choice composition of a sequential composition on the left and a rewrite rule on the right. The sequential composition is clearly problematic, as the first rewrite rule produces an expression incompatible with the second rule. Our type system will give the sequential composition a strategy type with an empty trace (\(\_ \rightarrow \_\)) which indicates that there is no possible execution path.

But, the overall strategy \(e_5\) will still work fine when using the rule on the right. Therefore, in this case the overall composition has the same type as the rule on the right: \(1 * a_0 \xrightarrow{[a]} a_0\). This shows that the composed strategy will work for expressions that are inputs for the rule on the right.

To alert the user of the problematic sequential composition, an implementation of our type system could issue a warning to the user:

```
let e5 = (rule m * n -> n * m ; rule m + n -> n + m) ||
// ^ Warning ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// This strategy is guaranteed to fail at runtime,
// but the overall strategy can still succeed.
    rule 1 * v -> v : 1 * a_0 \xrightarrow{[a]} a_0
```

In Section 5.3 we will see the typing rule \(T-S-App\) that could raise these warnings as soon as the set of traces in the conclusion becomes empty.

As for the behaviour of Rewrite \(S_t\) in this example, Rewrite \(S_t\) has the ability to assign an empty traced strategy type to the problematic sequential composition, but its rejection of empty traced types is delayed until typechecking the strategy execution (\(T-Run\) in Section 4.4.3), so Rewrite \(S_t\) can neither raise a warning nor an error for this example. Besides, in its formal representation, this example contains recursive types,
which are not supported by Rewrite $S_t$.

### 5.2.3 Problematic Strategy Compositions Causing Errors

In this section, we discuss examples of problematic strategies that are guaranteed to fail at runtime for all possible inputs and are, therefore, rejected by our type system.

**Example 5.7**

We consider a problematic sequential composition first:

```plaintext
let e6 = rule 2 * n -> n + n ; rule 2 + 3 -> 5
```

On a quick glance, this might seem like a valid composition, as the first rule produces an addition expression which is expected by the second rule. However, clearly we cannot instantiate the same variable $n$ with 2 and 3 at the same time. Formally in our type system this will be detected as there will be no trace for the unified type. This is a similar situation for the example before where we reported only a warning. Thus, to distinguish when raising a warning and when rejecting a program with an error, we analyze the context where an empty traced strategy is used. If it is used in a larger composition (as in Example 5.6) we will only raise a warning, but if we try to bind an empty traced strategy to a name using `let`, we will raise an error:

```plaintext
let e6 = rule 2 * n -> n + n : 2 * a0 [a] -> a0 + a0

// ^ Error
// This composition is guaranteed to fail at runtime,
// resulting in the failure of the overall strategy.
// There is no trace for the composed strategy type
// after unification of a0 + a0 and 2 + 3

rule 2 + 3 -> 5 : 2 + 3 [b] -> 5
```

In Section 5.3 we will see the typing rule T-Let that will fail with an error for empty traced strategies.

For this example, both Typed ELEVATE and Rewrite $S_t$ can detect the problem. If the rules are not defined in a fallible way, Typed ELEVATE will raise a type error because it cannot unify the closed variant types representing 2 and 3. Otherwise, this example will normally pass the typecheck in Typed ELEVATE as explained in Section 4.2. For Rewrite $S_t$, it can detect the problem and assign an empty traced type.
to this composition, but the error will not be raised until typechecking the execution of this strategy.

We can see a similar example in Section 4.3, which is also a sequential composition, and it can be expressed as follows.

\[
\text{let } \text{re2 = rule } m \ast \varnothing \rightarrow \varnothing ; \text{ rule } 1 \ast n \rightarrow n
\]

This composition obviously would not work, and both Rewrite \( S_t \) and Core \( S_t \) can detect this problem. In Core \( S_t \), an error is triggered for this example.

\[
\text{let } \text{re2 = rule } m \ast \varnothing \rightarrow \varnothing : a_0 \ast 0 \rightarrow 0
\]

\[
; \quad // \quad ^{\text{\small Error}}
\]

\[
// \quad \text{This composition is guaranteed to fail at runtime,}
// \quad \text{resulting in the failure of the overall strategy.}
// \quad \text{There is no trace for the composed strategy type}
// \quad \text{after unification of } 0 \text{ and } 1 \ast b_0
\]

\[
\text{rule } 1 \ast n \rightarrow n : 1 \ast b_0 \rightarrow b_0
\]

---

Example 5.8

Finally, we consider a more slightly complicated example with a sequential composition of two choice compositions:

\[
\text{let } e_7 = (\text{rule } m \ast n \rightarrow n \ast m | | \text{ rule } m + n \rightarrow n + m) ;
\]

\[
(\text{rule } v - 0 \rightarrow v | | \text{ rule } w / 1 \rightarrow w)
\]

Here no successful execution is possible, as the expressions produced by the first strategy must contain a multiplication or an addition which are incompatible with the second strategy expecting a subtraction or a division. We can give each choice composition a valid strategy type, but we can’t find a trace for the composition after unifying the output type of the first strategy with the input type of the second one.

\[
\text{let } e_7 = (\text{rule } m \ast n \rightarrow n \ast m | | \text{ rule } m + n \rightarrow n + m) :
\]

\[
a_0 \ast a_1 | \ b_0 + b_1 \rightarrow a_1 \ast a_0 | \ b_1 + b_0
\]

\[
; \quad // \quad ^{\text{\small Error}}
\]

\[
// \quad \text{This composition is guaranteed to fail at runtime,}
// \quad \text{resulting in the failure of the overall strategy.}
// \quad \text{There is no trace for the composed strategy type}
\]
Following a similar process, Rewrite $S_t$ can also detect the problem in this example. Actually, the first example in Section 4.3 is a similar sequential composition, involving a rewrite rule and a choice composition:

\[
\text{let } \text{re3 = } \text{rule } ? -\rightarrow 1 + z ; (\text{rule } 1 * n -\rightarrow n || \text{rule } \theta + m -\rightarrow m) \]

The tracing mechanisms in both Rewrite $S_t$ and Core $S_t$ can detect that no successful execution paths exist for this composition. In Core $S_t$, this composition triggers an error as follows.

\[
\text{let } \text{re3 = } \text{rule } ? -\rightarrow 1 + z : ? -\rightarrow 1 + a_0 ; \]

\[
// ^ \text{ Error}
// \text{This composition is guaranteed to fail at runtime,}
// \text{resulting in the failure of the overall strategy.}
// \text{There is no trace for the composed strategy type}
// \text{after unification of } 1 + a_0 \text{ and } 1 * b_0 | 0 + c_0
\]

\[
(\text{rule } 1 * n -\rightarrow n || \text{rule } \theta + m -\rightarrow m) : \]

\[
1 * b_0 | 0 + c_0 -\rightarrow b_0 | c_0
\]

5.2.4 Practical Examples

So far, we have deliberately discussed easy-to-follow examples rewriting arithmetic expressions in a simple expression language. To demonstrate the practical use of our type system, we discuss now a more realistic example taken from a practical compiler optimizations case study.

Example 5.9

Loop tiling is a traditional compiler optimization that improves the order of memory accesses resulting in better cache performance. Hagedorn et al. [2020] express loop tiling as a rewrite strategy over the functional array programming language RISE. We encode the syntax of RISE using our row-polymorphism structural types, but use a simplified notation here.

We give types to rewrite rules over functional RISE expressions, such as:
Chapter 5. Traced Types for Safe Strategic Rewriting

let mapFusion = rule Map f . Map g -> Map (f . g) :
   Map a₀ ◦ Map a₁ [a] → Map (a₀ ◦ a₁)

let mapFission = rule Map (f . g) -> Map f . Map g :
   Map (a₀ ◦ a₁) [a] → Map a₀ ◦ Map a₁

let transposeMove = rule Map (Map f) . T -> T . Map (Map f) :
   Map (Map a₀) ◦ T [a] → T ◦ Map (Map a₀)

let splitJoin32 = rule Map f -> Join . Map (Map f) . Split 32 :
   Map a₀ [a] → Join ◦ Map (Map a₀) ◦ Split 32

The mapFusion / mapFission rules merge or split two successive maps.

transposeMove moves a transposition of a matrix (T) before or after two nested maps.
Finally, splitJoin32 is the crucial rewrite for performing tiling, as it splits a single map (compiled into a loop) into two nested maps.

To compose these rules for performing tiling, we need to apply the rewrite strategies to specific sub-expressions. We define custom ad-hoc traversals, as introduced by Hagedorn et al. [2020]:

let id = rule x -> x : a₀ [a] → a₀

let function = st s =>
   rule f x -> (f , x) ; s && id ; rule (g , x) -> g x :
   (a₀ [a] → a₁) ⇒ (a₀ a₂ [a] → a₁ a₂)

let argument = st s =>
   rule f x -> (f , x) ; id && s ; rule (f , y) -> f y :
   (a₀ [a] → a₁) ⇒ (a₂ a₀ [a] → a₂ a₁)

let argOfMap = st s => rule Map f -> Map f ; argument s :
   (a₀ [a] → a₁) ⇒ (Map a₀ [a] → Map a₁)

let fmap = st s => function (argOfMap s) :
   (a₀ [a] → a₁) ⇒ (Map a₀ a₂ [a] → Map a₁ a₂)

The function traversal applies the given strategy s to the function f of a function application f x. This traversal is implemented by first deconstructing the function application syntax into a pair of its components. Then the both combinator && applies two strategies each to one element of a pair of expressions. Finally, the function application syntax is reconstructed. By using the identity strategy id we preserve the context while applying the strategy s to the desired sub-expression. Using the same technique, we define the argument traversal and building up on it, we define
argOfMap and fmap to apply a strategy to the argument of a fully applied map. Our current type system can express these ad-hoc traversals but not generic traversals found in many strategy languages, we discuss in Section 5.4.2 a possible extension using first-class labels that would allow generic traversals.

Using these ingredients we can define loop tiling below as originally described in Hagedorn et al. [2020]:

```ml
let tile2D = fmap (function splitJoin32)

function splitJoin32

loopInterchange2D :

Map

(Join ◦ Map (Map Join) ◦ Map T ◦ Map (Map (Map a0))) ◦

Map T ◦ Map (Map Split 32) ◦ Split 32) a2
```

The fairly complex type reflects the intuition behind two-dimensional tiling: two nested loops (represented here as functional maps) are rewritten into four nested loops requiring reshaping of the input data by splitting and transposition before transposing and rejoining the output data. The data reshaping will result in changes to the indexing in the generated imperative code, providing the desired performance benefit.

While the original paper presents a generic n-dimensional tiling, in our strategy language we are restricted to pick a fixed n, such as 2 here, as we do not support recursive strategies. We discuss in Section 5.4.2 our trade-offs, prioritizing strong formal guarantees and precise types over supporting recursion. We also discuss a possible future extension that could introduce (limited) forms of recursion without loosing all precision in the types.

In this section, we have seen a number of examples showing the types we assign to strategies and strategy combinators. We have seen successfully composed strategies as well as problematic ones that are highlighted with a warning or rejected with an error. Crucially, traces indicate the possible execution paths that can be taken at runtime by a strategy. Problematic strategies are identified when the set of traces is empty, indicating that no possible execution path exists. In the next section, we formalize these intuitions and formally show that well-typed but empty-traced strategies must fail at runtime, justifying why our type system rejects such strategies. Furthermore, we will prove that our type system is sound, and formally clarify what it means for well-typed and well-traced strategies to “not go wrong”.


5.3 Core \( S_t \), Formalized

In this section we present the formalization of Core \( S_t \). The meta-level syntax conventions used in this chapter are the same as those in Chapter 3 (see Section 3.3).

5.3.1 Term Syntax

Figure 5.1 shows the syntax of terms and patterns.

**Patterns** Patterns (denoted by \( p \)) are used in rewrite rule definitions, include pattern variables ranged over valid variable names (denoted by \( x, y, z, \cdots \)), unit pattern (\( () \)), pair pattern (\( (p, p) \)), and variant pattern (\( (\ell \ p) \)) where \( \ell \) is the label to be matched. For simplicity, all patterns in this paper are linear, namely each bound pattern variable only appears once in a pattern. The linearity requirement is enforced by typing rules for patterns in Figure 5.7.
Terms Terms (denoted by $e$) include variables ranged over valid variable names (denoted by $x,y,z,\cdots$). As the basic component of the language, rewrite rules can be defined with the lambda-like syntax ($\text{rule } p \rightarrow e$), where $p$ is the pattern to be matched or the LHS of the rule, and $e$ is the RHS of the rule constructed with the variables from $p$ and other syntax components. Thus, the term syntax provides unit constructor ($()$), pair constructor ($([e_m,e_n])$), and variant constructor or label application ($\ell \ e$) for use inside rule definitions. It is straightforward to see that the syntax of these terms is similar to that of patterns and a conversion $p2e$ from patterns to terms can be easily defined (see Figure 5.2). There are three constant strategy combinators, seq for sequential composition, choice for the (non-deterministic) choice composition of strategies, and both for the pairing composition of strategies. To allow the programmers to define strategy combinators by themselves, the term syntax also provides strategy combinator abstraction ($\text{st } x \Rightarrow e_b$) where $x$ is the bound variable representing a strategy and $e_b$ is the body of the combinator definition, and strategy combinator application ($e_f \leftarrow e$) where $e_f$ is the strategy combinator and $e$ is the argument strategy. To get the execution result of strategies, one can apply an input $e_i$ to a strategy $e_s$ by writing strategy execution ($e_s \leftarrow e_i$), and the execution result can either be (succ $e$) where $e$ is the successfully rewritten expression, or simply fail. To support the choice combinator and the both combinator, the non-deterministic choice between two results ($e_m \sqcap e_n$) and the pair of two results ($e_m \sqcap e_n$) are also involved in the syntax, but in most cases, programmers do not need to use these operators directly. Finally, the term syntax provides let-bindings ($\text{let } x = e_f \text{ in } e$) for polymorphic abstractions of strategies, strategy combinators and execution results.

Informal Presentation of Terms Shown below is a term written in the formal syntax. It contains a strategy combinator definition (swapOps from Example 5.4) and strategy applications.

```plaintext
let swapOps = st s => seq ≔ (rule Op (op, (m, n)) → Op (op, (n, m))) ≔ s
  in (swapOps ≔ (rule Op (Mul (), (1 (), v)) → v))
    \ succ (Op (Mul (), (2 (), 1 ()))))
```

By applying the syntactic sugar for Core $S_t$ terms such as numbers and binary operators in rewrite rule definitions, we get a term in the simplified syntax used in Section 5.2.

```plaintext
let swapOps = st s => (rule m op n → n op m) ; s
  in swapOps (rule 1 * v → v) (2 * 1)
```
### 5.3.2 Type Syntax

Figure 5.3 shows the syntax of traces, kinds, and types in Core \( S_t \), together with the syntax of the kinding, tracing and typing environments.

<table>
<thead>
<tr>
<th>Trace Variables ( \gamma )</th>
<th>( \alpha \mid \beta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Identifier Sets ( \varphi )</td>
<td>( [(\alpha_i)_{i \in N}] )</td>
</tr>
<tr>
<td>Trace Member Sets ( \phi )</td>
<td>( [(\beta_i)_{i \in N}] )</td>
</tr>
<tr>
<td>Trace Variable Sets ( \psi )</td>
<td>( [(\gamma_i)_{i \in N}] )</td>
</tr>
<tr>
<td>Traces ( \alpha \rhd \phi )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kinds ( \kappa )</th>
<th>( \mathcal{R} \mid \mathcal{T} \mid \mathcal{T}_* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{R} )</td>
<td>( {(l_i)_{i \in N}} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types ( t )</th>
<th>( \tau \mid \rho \mid \omega )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau )</td>
<td>( v \mid {\ell : \tau } \mid (\nu \text{ as } \rho) \mid (\nu) \mid (\tau_m, \tau_n) )</td>
</tr>
<tr>
<td>( \rho )</td>
<td>( \nu \mid \ell : \tau \mid \rho )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( \varphi \mid \tau \mid \tau_p \xrightarrow{\varphi} \tau_e \mid (\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schemes ( \bar{\sigma} )</th>
<th>( \text{tr } \Phi \cdot \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma )</td>
<td>( \forall (v : \kappa) \cdot \sigma \mid \omega )</td>
</tr>
</tbody>
</table>

| Kinding Env. \( \Delta \) | \( \Delta, v : \kappa \) |
| Typing Env. \( \Gamma \) | \( \Gamma, x : \bar{\sigma} \) |
| Tracing Env. \( \Phi \) | \( \Phi, \alpha \rhd \phi \) |
| Rule Env. \( \Theta \) | \( \star \mid \Theta, x : \tau \) |
| Strategy Combinator Env. \( \Sigma \) | \( \Sigma, x : \omega \) |

Figure 5.3: Syntax of traces, kinds, types and type schemes and kinding, tracing and typing environments.
Traces  Being similar to Rewrite $S_t$, an important part of the type system of Core $S_t$ is tracing. With traces, we are able to observe each possible way of rewriting at the type level, but in Core $S_t$, traces are encoded in a less verbose way. Traces are constructed with trace variables ($\gamma$) which can be partitioned into two subsets: trace identifiers ($\alpha$) and trace members ($\beta$). We use different naming conventions for different kinds of trace variables in the examples, and different meta variables to represent them in our formalism for convenience and clarity. Notice however that it is not necessary for trace variables to have any syntactical difference because the tracing environment can distinguish them. The sets (of distinct elements) formed by different group of trace variables are also denoted by different meta variables: $\varphi$ for trace identifier sets, $\phi$ for trace member sets, and $\psi$ for trace variable sets. Finally, we are able to define traces. Each trace denoted as $\alpha \triangleright \phi$ is identified by the trace identifier $\alpha$ and the trace member set $\phi$ contains the remaining trace variables of this trace. Besides adapting common set operations, for convenience, the following conversion rules are used for writing the sets mentioned above. We show the rules for trace identifier sets as an example, and the same rules can be applied to the other two kinds of sets.

$$[\alpha, \varphi] = [\alpha, (\alpha_i)_{i \in N}] \text{ if } \varphi = [(\alpha_i)_{i \in N}] \text{ and } \alpha \notin \varphi$$

$$[\varphi_a, \varphi_b] = [(\alpha_i)_{i \in N}, (\alpha_j)_{j \in M}] \text{ if } \varphi_a = [(\alpha_i)_{i \in N}] \text{ and } \varphi_b = [(\alpha_j)_{j \in M}] \text{ and } \varphi_a \cap \varphi_b = \emptyset$$

Kinds and Types  There are three syntactical categories of types together forming the type universe $t$ in Core $S_t$: traceable types ($\tau$), row types ($\rho$), and traced types ($\omega$). Traceable types ($\tau$) are of kind $T \cdot$, and they include type variables ($\nu$), variant types ($\langle \rho \rangle$) constructed from rows, equi-recursive variant types ($\nu$ as $\langle \rho \rangle$) where $\nu$ is a variable representing the type itself, unit type ($\langle \rangle$), and pair types ($\langle \tau_m, \tau_n \rangle$). For each case in the definition of $\tau$, there is a trace variable set (or trace member set) optionally attached to it at the subscript position. This is how tracing works at the syntactical level: instead of being deeply tied to the underlying types, traces are another layer of type-level information overlaid on the underlying types. The following conversion rule is used for folding and unfolding equi-recursive variant types, where $\text{erase}$ is a function erasing all traces and defined in Figure 5.4.

$$\nu \text{ as } \langle \rho \rangle \simeq \langle \rho \rangle[\nu \mapsto \text{erase}(\nu \text{ as } \langle \rho \rangle)]$$

Since all traces in the folded type are erased during unfolding, the traces stay the same before and after this conversion. If the case for the equi-recursive variant types
**Chapter 5. Traced Types for Safe Strategic Rewriting**

110

 erase : t → t
 erase(τφ) = erase(τ)
 erase(ν) = ν
 erase(⟨ρ⟩) = ⟨erase(ρ)⟩
 erase(ν as ⟨ρ⟩) = ν as ⟨erase(ρ)⟩
 erase(()) = ()
 erase((τm, τn)) = (erase(τm), erase(τn))
 erase(·) = ·
 erase(ℓ : τ | ρ) = (ℓ : erase(τ) | erase(ρ))
 erase(φ ▶ τ) = [] ▶ erase(τ)
 erase(τp ▷ τe) = erase(τp) ▷ erase(τe)
 erase((τp ▷ τe) ⇒ ω) = erase(τp ▷ τe) ⇒ erase(ω)

Figure 5.4: Definition of trace erasure

is absent from subsequent definitions, it means that all equi-recursive variant types are unfolded in advance.

Row types (ρ) are sequences of label-type pairs ℓ : τ ending with row variables (ν) or empty row (·), where the order of the label-term pairs is insignificant, and labels are all distinct. The set of all possible labels and the set of all possible variable names are disjoint. Rows are differentiated from ordinary types by their kinds, named row kinds (R), which are sets of labels that are not present in the rows. Compared with Typed ELEVATE and Rewrite S_t, Core S_t uses a simpler form of row kinds. For a row variable, its kind describes the finite set of labels that the instantiation of the row variable must not contain [Blume et al., 2006; Hillerström and Lindley, 2016]. Although rows are used to constructed traceable types, row types themselves are not traceable. This indicates that we can switch between different representations of rows, or even different specifications of structural types in general, and add the tracing mechanism to it.

Traced types (ω) contains strategy result types (φ ▶ τ), strategy types (τp ▷ τe), and strategy combinator types ((τp ▷ τe) ⇒ ω). Traced types and traceable types together form the T kind, which is the super kind of T. As its name suggests, each
case of the traced types carries one or more trace identifier sets $\varphi$, identifying the traces in the corresponding traceable types. The strategy result type $\varphi \triangleright \tau$ is the type of the result of strategy executions which could be either $\text{succ } e$ or $\text{fail}$. In the $\text{succ } e$ case, $\tau$ in $\varphi \triangleright \tau$ gives the type of $e$. Thus, the strategy result type can be considered as an analog of the $\text{Maybe a}$ type in Haskell. The strategy type $\tau_p \triangleright \tau_e$ is the type of strategies which are either individual rewrite rules or large strategies composed from smaller strategies. For a single rewrite rules, the type can be read as "a rule rewriting a term of type $\tau_p$ to a term of type $\tau_e$", and for more general strategies, the traces in $\tau_p$ and $\tau_e$ can be more complex and each trace marks one possible way of rewriting. The strategy combinator types $(\tau_p \triangleright \tau_e) \Rightarrow \omega$ is the type of strategy combinators which are either provided by the language as primitives such as $\text{seq}$, $\text{choice}$ and $\text{both}$, or defined by programmers using the $\text{st}$ keyword. A subtlety here is that instead of directly taking a strategy of type $\tau_p \triangleright \tau_e$ as its argument, a strategy combinator of type $(\tau_p \triangleright \tau_e) \Rightarrow \omega$ expects a strategy whose type is compatible with $\tau_p \triangleright \tau_e$ and return a traced type by applying the result of trace computations to $\omega$. More about the trace computation will be introduced in Section 5.3.5. Finally, the trace identifier sets in traced types are written out only for clarity, and they can actually be computed using the tracing rules which will be introduced in Section 5.3.3.

**Informal Presentation of Types** The expression (rule 1 * v -> v || rule 2 * w -> w) (from Example 5.1) is used here as an example to discuss how the formal types relate to the informal presentation used in Section 5.2. The formal term syntax for the expression is:

\[
\text{choice} \leftarrow \text{rule } Op (\text{Mul } (),(1 (),v)) \rightarrow v \right)
\]

\[
\larrow \text{rule } Op (\text{Mul } (),(2 (),w)) \rightarrow Op (\text{Add } (),(w,w)))
\]

Its formal type is as follows:

\[
\langle Op : (\langle Mul : ()_{[b,c]} | v_a \rangle, (1 : ()_{[b]} | 2 : ()_{[c]} | v_c),
\]

\[
(v \text{ as } \langle Op : (\langle Add : () | v_e, (v,v) | v_d)_{[b_0,c_0]} \rangle) | v_b \rangle \rightarrow [b,c]
\]

\[
\langle Op : (\langle Add : ()_{[c]} | v_e \rangle, (v \text{ as } \langle Op : (\langle Add : () | v_e, (v,v) | v_d)_{[c_0]} \rangle),
\]

\[
(v \text{ as } \langle Op : (\langle Add : () | v_e, (v,v) | v_d)_{[c_0]} \rangle) | v_d \rangle_{[b_0]}
\]

There are two trace identifiers, $b$ and $c$, corresponding to the two rules composed by the $\text{choice}$ combinator. For demonstration, we focus on the types traced by trace
variables identified by $b$, which are $b$ and $b_0$. By performing a bottom-up traversal of the whole type while ignoring the $c$ trace, we get the following type. Later in Section 5.3.5, we can see that the selection of a specific trace (such as the one identified by $b$ here) can be formalised as a Select operator.

\[
\langle\text{Op}: (\langle\text{Mul}: ()_{[b]} | v_a\rangle, \langle 1 : ()_{[b]} | 2 : () | v_c\rangle, (v \text{ as } \langle\text{Op}: (\langle\text{Add}: () | v_e\rangle, (v, v)) | v_d\rangle)_{[b]\backslash b})\rangle \overset{[b]}{\rightarrow}
\langle\text{Op}: (\langle\text{Add}: () | v_e\rangle, (v \text{ as } \langle\text{Op}: (\langle\text{Add}: () | v_e\rangle, (v, v)) | v_d\rangle)_{[b]\backslash b})\rangle
\]

By applying the conversion rule for equi-recursive types (which exist here because of type unifications), we get a simpler type where irrelevant parts are greyed out.

\[
\langle\text{Op}: (\langle\text{Mul}: ()_{[b]} | v_a\rangle, \langle 1 : ()_{[b]} | 2 : () | v_c\rangle, (v \text{ as } \langle\text{Op}: (\langle\text{Add}: () | v_e\rangle, (v, v)) | v_d\rangle)_{[b]\backslash b})\rangle \overset{[b]}{\rightarrow}
\langle\text{Op}: (\langle\text{Add}: () | v_e\rangle, (v \text{ as } \langle\text{Op}: (\langle\text{Add}: () | v_e\rangle, (v, v)) | v_d\rangle)_{[b]\backslash b})\rangle
\]

For this type, we can already see its connection with the expression rule $\text{Op} (\text{Mul} (), (1 ()_{v}) \rightarrow v$, and how the type representing a single rewrite path can be extracted with the assistance of the trace variables. When using syntactical sugar for numbers, binary operators and row variables, the presentation of the type becomes more clear:

\[
\langle\langle 1_{[b]} | 2 \rangle \rangle \langle\ast_{[b]} \rangle (v \text{ as } \langle v \langle + | \rangle v \rangle)_{[b]\backslash b}) \overset{[b]}{\rightarrow} (v \text{ as } \langle v \langle + | \rangle v \rangle)_{[b]\backslash b})
\]

We can also see here the correspondence between the trace members and the pattern variables. Now we can drop the underlying structural types and the gray irrelevant parts and get a simplified type $1 * b_0 \overset{[b]}{\rightarrow} b_0$, which still clearly shows the rewriting performed by the corresponding rule. We can do the same for the other trace identified by $c$, and we will get another simplified type $2 * c_0 \overset{[c]}{\rightarrow} c_0 + c_0$. Putting these two types together and separating the parts identified by $b$ and $c$ with vertical bars, we finally get the informal type as originally shown in Section 5.2:

\[
1 * b_0 \mid 2 * c_0 \overset{[b,c]}{\rightarrow} b_0 \mid c_0 + c_0
\]

**Type schemes** The type system of Core $S_t$ is also equipped with generalized types. A type scheme ($\overline{\sigma}$) represents an universally quantified type, and carries a tracing
environment Φ, to be instantiated with fresh traces. Type schemes without tracing
environment are denoted as σ and inductively defined in a standard way. The kind of
the bound type variables are specified at the binding site. The (∀(α_i : κ_i)_{i ∈ N}) syntax
may be used in the text to collectively describe a series of quantifiers when the order
is insignificant.

**Environments**  Figure 5.3 gives the definitions of various environments. The kind-
inging (Δ) and typing environments (Γ) are standard: they can be either empty (·) or
extended with a type variable and its kind (ν : κ) or a variable and its type scheme
(x : σ). The tracing environment (Φ) is a collection of traces, and it provides all the
tracing information required by other environments and types. The following con-
version rule is used if all the traces in the environment do not contain trace members.

\[ (α_i ▷ [[]])_{i ∈ N} = φ ▷ [[]] \text{ if } φ = [(α_i)_{i ∈ N}] \]

The rule environment (Θ) is the typing environment used while defining a rewrite
rule, and it contains the variable-type pairs extracted from the LHS of the rule. It
is possible that a rule is defined without any variables on the LHS, so to clearly distin-
guish the inner and outer sides of rules, a rule environment can be inductively
constructed from an empty environment (·) or a nameless placeholders (★) which
make the rule environment non-empty without introducing variables. The strategy
combinator environment (Σ) is used while defining a strategy combinator, and it only
holds variables of the strategy type. By the way, the erase function for erasing the
traces in types can be easily generalized to be used on environments.

**Linearity of Strategy Variables**  Our goal of designing this type system is to pre-
cisely reflect how strategies manipulate the rewritten programs, but some strategy
definitions are very tricky to give a precise type. Let us consider, the challenging
strategy combinator \( st \ x ⇒ seq \) that sequentially composes a strategy \( x \) with itself. For assigning a type to this strategy combinator, we only know that
\( x \) has a general strategy type \( τ_p ⇔ τ_e \). As \( x \) is sequentially composed with itself,
its output type becomes its own input type, and therefore, its output type \( τ_e \) must
match its input type \( τ_p \). Via unifications, the type of \( x \) (and its sequential composi-
tion with itself) becomes the imprecise strategy type \( τ_q ⇔ τ_q \). Following this pro-
cedure, we can arbitrarily extend the chain of sequential composition, for example,
\( st \ x ⇒ seq \) becomes \( (seq \ ⇔ x ⇔ x) ⇔ x \), all resulting in the same imprecise type. This is
clearly in conflict with our goal of precise typing. To avoid this problem, we restrict
the repeated use of strategy variables, such as $x$ in the example, with linearity, that is,
a strategy variable can only be used once in a strategy combinator definition. Besides,
due to the same reason, recursive strategy combinator definitions are not supported
by our type system. To enforce linearity we remove the contraction structural rule
from the tracing environment and the strategy combinator environment, enforcing
that there cannot be repeated occurrences of the same trace identifier or the same
strategy variable. More details about typing strategy combinators will be covered in
Section 5.3.5. As a workaround to avoid this problem, we can simply bind a strategy
to a name via `let` which allows the sequential composition of a strategy with itself
in a safe way, as described in Section 5.3.6.

**Typing Judgement** Typing judgements are of the form $\Delta; \Gamma; \Phi \downarrow; \Theta \vdash e : t \downarrow; \Phi \uparrow; \Sigma \uparrow$, stating that the term $e$ has type $t$ under a series of input environments and the typing judgement produces environments as its output. The equivalence of types in this paper is up to alpha-renaming. If the name of an input environment collides with an output environment, the $\downarrow$ and $\uparrow$ superscripts are used to distinguish the input one and the output one, respectively. For the input environments on the left of the turnstile, $\Delta$ provides the kinding information, and it ensures that $t$ and the types in all other environments (including those output environments) are well-kinded; $\Gamma$ provides the type information for variables bound by `let`; $\Phi \downarrow$ is the input tracing environment, and it ensures that the types in $\Sigma \downarrow$ and $\Theta$ are well-traced; $\Sigma \downarrow$ holds the input type of variables bound by `st`; $\Theta$ holds the type of variables bound by `rule`. For the output environments on the right of the reverse turnstile, $\Phi \uparrow$ is the output tracing environment, and it ensures that $t$ and all types in $\Sigma \uparrow$ are well-traced; $\Sigma \uparrow$ holds the output type of variables bound by `st`. If any of the environment must be empty, we omit it from the typing judgement. For example, $\Delta; \Gamma; \Phi \downarrow; \Sigma \downarrow; \Theta \vdash e : t \downarrow; \Phi \uparrow; \Sigma \uparrow$ is equivalent to $\Delta; \Gamma; \Phi \downarrow; \Sigma \downarrow; \vdash e : t \downarrow; \Phi \uparrow; \Sigma \uparrow$. If the output environments are exactly the same as their input counterparts, we omit them together with the reverse turnstile. For example, $\Delta; \Gamma; \Phi \downarrow; \Sigma \downarrow; \Theta \vdash e : t$ is equivalent to $\Delta; \Gamma; \Phi \downarrow; \Sigma \downarrow; \vdash e : t \downarrow; \Phi \uparrow; \Sigma \uparrow$.

**5.3.3 Well-formedness of Types**

Figure 5.5 shows the kinding rules checking the well-formedness of types. Kinding judgments are of the form $\Delta; \Phi \vdash t : \kappa$, stating that the type $t$ is well-formed and has kind $\kappa$ in the kinding environment $\Delta$ and the tracing environment $\Phi$. When the trac-
\[ \frac{\nu : \kappa \in \Delta}{\Delta \vdash \nu : \kappa} \quad \text{K-Var} \]
\[ \frac{\Delta \vdash () : \tau \bullet}{\Delta \vdash (\tau_m, \tau_n) : \tau \bullet} \quad \text{K-Pair} \]
\[ \frac{\Delta \vdash \rho : \mathcal{R}}{\Delta \vdash \langle \rho \rangle : \tau} \quad \text{K-Unit} \]
\[ \frac{\Delta \vdash \nu : \tau \bullet \Delta \vdash \rho : \mathcal{R}}{\Delta \vdash \nu \; \text{as} \; (\rho) : \tau \bullet} \quad \text{K-Recursive} \]
\[ \frac{\Delta \vdash \tau : \tau \bullet \Delta \vdash \nu : \tau \bullet}{\Delta \vdash (\tau \nu) : \tau \bullet} \quad \text{K-Pair} \]
\[ \frac{\Delta \vdash \rho : \mathcal{R} \quad \Delta \vdash \tau : \tau \bullet \ell \in \mathcal{R} \quad \mathcal{R}' = \mathcal{R} \setminus \ell}{\Delta \vdash (\ell : \tau | \rho) : \tau \bullet} \quad \text{K-RowExt} \]
\[ \frac{\Delta \vdash \tau_p : \tau \bullet \quad \Delta \vdash \tau_e : \tau \bullet \quad \Phi \vdash \tau_p \phi \quad \Phi \vdash \tau_e \phi}{\Delta ; \Phi \vdash \tau_p \phi \rightarrow \tau_e : \tau} \quad \text{K-Strategy} \]
\[ \frac{\Delta \vdash \tau : \tau \bullet \quad \Phi \vdash \tau + \phi}{\Delta ; \Phi \vdash \tau + \phi : \tau} \quad \text{K-Result} \]
\[ \frac{\Delta ; \Phi \vdash \tau_p \phi \rightarrow \tau_e : \tau \quad \Delta ; \Phi \vdash \omega : \tau}{\Delta ; \Phi \vdash (\tau_p \phi \rightarrow \tau_e) \Rightarrow \omega : \tau} \quad \text{K-Combinator} \]

Figure 5.5: Kinding rules

The kinding environment is not used, we omit it from the judgement. Most of the kinding rules are straightforward. Most row kinding rules here are similar to those in [Hillerström and Lindley, 2016], except the K-\text{Variant} rule, which is more relaxed in our type system. Instead of requiring the row kind to be empty to construct a variant type, we do not have this requirement and the kind of \( \rho \) in K-\text{Variant} can be arbitrary. This still ensures that there are no repeated labels in a row, and also gives us the flexibility of expressing the certain absence of some labels in a row as in [Blume et al., 2006].

Apart from this, there are three specific kinding rules for our type system: K-\text{Strategy}, K-\text{Result}, and K-\text{Combinator}. They are the kinding rules for traced types, checking for the well-formedness of both the traces and the underlying types. Thus, a set of tracing rules is required to define them.

Figure 5.6 shows the tracing rules which check the well-formedness of traces and also compute the trace identifier set for types. Tracing judgements are of the form \( \Phi \vdash \tau \phi \), stating that the type \( \tau \) is well-traced in the tracing environment \( \Phi \) and the corresponding trace identifier set for \( \tau \) is \( \phi \). The most important tracing rule is Tr-\text{Set}. For any traceable type \( \tau \) traced by a trace variable set \( \{ \gamma, \psi \} \), we firstly check the well-formedness of \( \tau \) traced by a single trace variable \( \gamma \), which will be handled
by either Tr-Member or Tr-Identifier depending on $\gamma$, and we get a trace identifier set $\varphi_h$, which identify $\gamma$ and the trace variables inside $\tau$. Since the internal traces in $\tau$ are already checked, we use a unit type to replace $\tau$ and compute the trace identifier set $\varphi_t$ of $()_\psi$, and then give the final result $[\varphi_h, \varphi_t]$. For the Tr-Member rule, we use trace members to trace the type of pattern variables which are terminals in the syntax. In other words, it is impossible for (the type of) a variable to have any internal (tracing) structure (identified by the same trace identifier). Thus, for $\tau_\beta$, $\beta$ must be identified by a variable $\alpha$ which is distinct from the elements in the trace identifier set of $\tau$. For the Tr-Identifier rule, the unit pattern and the unit type is the only non-variable terminal in the pattern and type syntax, respectively, so the unit type is the only type that can be directly traced by trace identifiers. Furthermore, there are tracing rules closely related with the structure of patterns (and the terms that can be matched by patterns). Both elements of a pair pattern must be present simultaneously, so the Tr-Pair require both elements of the pair type to have the same trace identifier set. Only one label can be present in a variant pattern, so the Tr-Variant rule require the trace identifier set of the type associated with each label $\ell$ to be distinct from the rest of the row. The rest of the tracing rules are straightforward.
### 5.3.4 Typing Rules for Rewrite rules

Figure 5.7 shows the typing rules for rewrite rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **T-R-Lam** | fresh $\alpha$  
$\Delta \vdash \alpha : \tau_p \rightarrow \Phi; \Theta$  
$\Delta; \Gamma; \Phi; \Theta \vdash e : \tau_e$  
$\Delta; \Gamma; \Phi; \Theta ; \Sigma ; \Gamma ; \Phi ; \Theta \vdash rule \ p \rightarrow e \ : \ [\alpha] \rightarrow \tau_e \rightarrow \Phi$ |
| **P-Var** | fresh $\beta$  
$\Phi = \alpha \triangleright \{ [\beta] \}$  
$\Theta = \star, x : \tau_{[\beta]}$  
$\Delta \vdash x : \tau_{[\beta]} \rightarrow \Phi; \Theta$ |
| **P-Unit** | $\Phi = \alpha \triangleright []$  
$\Delta \vdash () \rightarrow \Phi; \Theta$ |
| **P-Pair** | $\Delta \vdash \alpha : \tau \rightarrow \Phi; \Theta$  
$\Delta \vdash \ell p : (\ell : \tau | \rho) \rightarrow \Phi; \Theta$ |
| **P-Label** | $\Delta \vdash \lambda x : \tau \in \Theta \rightarrow \Phi; \Theta$  
$\Delta \vdash \alpha \triangleright \phi$  
$\Theta \neq \cdot$  
$\Delta \vdash \ell \rightarrow (\ell : \tau | \rho) \rightarrow \Phi; \Theta$ |
| **T-R-Var** | $\Delta \vdash \ell p : (\ell : \tau | \rho) \rightarrow \Phi; \Theta$ |
| **T-R-Unit** | $\Delta \vdash \ell \rightarrow (\ell : \tau | \rho) \rightarrow \Phi; \Theta$ |
| **T-R-Pair** | $\Delta \vdash e : \tau \rightarrow \Phi; \Theta$ |

**Figure 5.7: Typing rules for rewrite rules**

**Typing Rule for Rewrite Rule Definitions**  
We start with the top rule **T-R-Lam** for typing rewrite rule definitions. A fresh trace identifier $\alpha$ is assigned to this rule. A rule environment $\Theta$ (together with its tracing environment $\Phi$) is computed from the pattern $p$ or the LHS of the rewrite rule, and then $\Theta$ is used for typing the RHS of the rewrite rule $e$. 
Pattern Typing Rules for the LHS  The typing judgement for patterns is of the form $\Delta \vdash \alpha p : \tau \downarrow \Phi; \Theta$, stating that given the kinding environment $\Delta$ and the assigned trace identifier $\alpha$, the pattern $p$ has type $\tau$, and the judgement produces a tracing environment $\Phi$ and a rule environment $\Theta$. The only trace in the tracing environment $\Phi$ is identified by $\alpha$, and $\Phi$ ensures that $\tau$ and all types in $\Theta$ are well-traced. As for the pattern typing rules, $P$-$\text{Var}$ assigns a fresh trace member $\beta$ to the underlying type $\tau$ of the pattern variable $x$, and then extend a non-empty rule environment with $x : \tau[\beta]$ as its output; $P$-$\text{Unit}$ simply takes the assigned trace identifier and produces a non-empty rule environment as its output; $P$-$\text{Pair}$ ensures the pattern is linear by checking if the output environments from both elements do not intersect; $P$-$\text{Label}$ gives a variant type containing the matched label $\ell$ and the corresponding type $\tau$, and according to the tracing rule, $\rho = \text{erase}(\rho)$.

Typing Rules for RHS  The remaining typing rules in Figure 5.7 are for typing the RHS of rewrite rules, and are straightforward. The condition $\Theta \neq \cdot$ is for prohibiting the usage of other language constructs in the body of rewrite rules. Noticeably, the typing rules for both sides of a rewrite rule definition are similar except the linear restriction for patterns, also sharing the same syntax. The reason for this is that the RHS of a rewrite rule can be matched against the LHS of another rewrite rule during compositions, and this similarity makes them comparable with each other.

Typing a Rewrite Rule  Figure 5.8 shows the actual typing derivation of a rewrite rule $(\text{rule } 1 * v \rightarrow v)$ taken from Example 5.1. The variable $v$ is renamed as $x$ for a more readable presentation. This typing derivation demonstrates how the pattern variable $x$ is captured by the rule environment and used by the RHS, and how the trace variables are attached to the underlying types.
$\Delta \models_{\alpha} () : () \vdash (\alpha \triangleright []) : \star$

\[\begin{align*}
\Delta \models_{\alpha} \text{Mul} () : \langle \text{Mul} : () \mid v_0 \rangle \vdash (\alpha \triangleright []) : \star
\end{align*}\]

\[\begin{align*}
\Delta \models (\text{Mul} (), (1 (), x)) : (\langle \text{Mul} : () \mid v_0 \rangle, (\langle 1 : () \mid v_1 \rangle, \tau^x_{[\beta]})) + (\alpha \triangleright [\beta] ; (\star, x : \tau^x_{[\beta]}))
\end{align*}\]

\[\begin{align*}
\text{fresh } \alpha
\end{align*}\]

\[\begin{align*}
\Phi = \alpha \triangleright [\beta] \quad \Theta = \star, x : \tau^x_{[\beta]} \quad \tau_p = \langle \text{Op} : (\langle \text{Mul} : () \mid v_0 \rangle, (\langle 1 : () \mid v_1 \rangle, \tau^x_{[\beta]})) \mid v_2 \rangle
\end{align*}\]

Figure 5.8: Typing derivation of a rewrite rule
5.3.5 Typing Rules for strategy Combinators

Figure 5.9 shows the typing rules for strategy combinators, including the typing rules for basic strategy combinators, strategy combinator definitions and strategy combinator applications.

**Typing Rules for Basic Combinators** The typing rules for the basic strategy combinators, seq, choice and both are straightforward. For seq, it sequentially composes two strategies, so it requires that the two strategies have the same trace and the output of the first strategy matches the input of the second one. For choice, it makes a nondeterministic choice between two strategies, so it requires that the two strategies have distinct traces, but their underlying structural type must be the same, and the two traces will be put together in the result type of the composition. For both, it composes two strategies to apply them simultaneously on a pair, so it requires that the two strategies have the same trace and their input and output types are paired respectively.

**Typing Rules for Custom Strategy Combinators** To construct strategy combinators, we introduced the st keyword, and the corresponding typing rule is T-S-Lam. According to the type rules for this type system, we distinguish the input and output environments, and only the output environments contain the final tracing information (after all trace computations). Thus, for the expression st x ⇒ eb, we have to extract the type of x from the output strategy combinator environment to ensure that the type precisely reflect the usage of x in eb. Consequently, this requires a carefully designed type for x in the input strategy combinator environment. In the T-S-Lam rule, x is assigned an input type with a fresh trace identifier and two fresh trace members on the LHS and the RHS of the strategy type, respectively. Besides, the underlying type of x always stay the same in the input and output environments, so it is directly used without any change. For this input type of x, an intuitive interpretation is that the type does not assume anything about tracing, and it has the most general trace, so when x is (linearly) used and involved in trace computations, the output type will capture the tracing information from the context. Formally, the correctness of the T-S-Lam rule is justified by Lemma 5.3.6 which shows that beta-reductions for strategy combinators preserve types.

The rule T-S-Var is for the usage of the bound variable in the body of a strategy combinator definition. It takes the type of the variable x from the input strategy
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Seq</td>
<td>$\text{fresh } \alpha, \beta_i, \beta_j, \beta_k \quad \Phi = \alpha \triangleright [\beta_i, \beta_j, \beta_k]$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma; \Phi; \Sigma \vdash \text{seq} : (\tau^i_{[\beta_i]} \xrightarrow{[\alpha]} \tau^j_{[\beta_j]} \Rightarrow (\tau^j_{[\beta_j]} \xrightarrow{[\alpha]} \tau^k_{[\beta_k]} \Rightarrow (\tau^i_{[\beta_i]} \xrightarrow{[\alpha]} \tau^k_{[\beta_k]})) + \Phi$</td>
</tr>
<tr>
<td>T-Choice</td>
<td>$\text{fresh } \alpha, \beta_i, \beta_j, \beta_k \quad \Phi = \alpha \triangleright [\beta_i, \beta_j, \beta_k]$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma; \Phi; \Sigma \vdash \text{choice} : (\tau^p_{[\beta_m]} \xrightarrow{[a_m]} \tau^e_{[\beta_n]} \Rightarrow (\tau^p_{[\beta_j]} \xrightarrow{[a_j]} \tau^e_{[\beta_k]} \Rightarrow)$</td>
</tr>
<tr>
<td></td>
<td>$((\tau^m_{[\beta_m]} \rightarrow (\tau^f_{[\beta_j]} \rightarrow \tau^k_{[\beta_k]})))) + \Phi$</td>
</tr>
<tr>
<td>T-Both</td>
<td>$\text{fresh } \alpha, \beta_i, \beta_j, \beta_k \quad \Phi = \alpha \triangleright [\beta_i, \beta_j, \beta_k]$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma; \Phi; \Sigma \vdash \text{both} : (\tau^m_{[\beta_m]} \xrightarrow{[a]} \tau^n_{\beta_n} \Rightarrow (\tau^j_{[\beta_j]} \rightarrow \tau^k_{[\beta_k]} \Rightarrow)$</td>
</tr>
<tr>
<td></td>
<td>$((\tau^m_{[\beta_m]} \rightarrow (\tau^f_{[\beta_j]} \rightarrow \tau^k_{[\beta_k]})))) + \Phi$</td>
</tr>
<tr>
<td>T-Lam</td>
<td>$\omega \cdot x = \text{erase}(\tau_m)<em>{[\beta_m]} \xrightarrow{[a]} \text{erase}(\tau_n)</em>{[\beta_n]}$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma; \Phi, \Sigma^\top, x : \omega \cdot x \vdash e_b : \omega \cdot \Phi^\top_i \Sigma^\top, x : \tau_m \rightarrow \tau_n$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma; \Phi, \Sigma^\top \vdash \text{st } x \Rightarrow e_b : (\tau_m \rightarrow \tau_n) \Rightarrow \omega \cdot \Phi^\top_i \Sigma^\top$</td>
</tr>
<tr>
<td>T-Var</td>
<td>$\omega = \tau_m \xrightarrow{[a]} \tau_n \quad \alpha \triangleright \phi \in \Phi^\top_i \quad x : \omega \in \Sigma^\top$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma; \Phi, \Sigma^\top \vdash x : \omega \cdot \alpha \triangleright \phi_i, x : \omega$</td>
</tr>
<tr>
<td>T-App</td>
<td>$\Delta; \Gamma; \Phi, \Sigma^\top \vdash e_f : (\tau_m \xrightarrow{\psi} \tau_n) \Rightarrow \omega_f \rightarrow \Phi^\top_i \Sigma^\top_f$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma; \Phi, \Sigma^\top \vdash e : \tau_j \xrightarrow{\psi} \tau_k \Rightarrow \Phi^\top_e \Sigma^\top_e$</td>
</tr>
<tr>
<td></td>
<td>$\text{erase}(\tau_m \xrightarrow{\psi} \tau_n) = \text{erase}(\tau_j \xrightarrow{\psi} \tau_k)$</td>
</tr>
<tr>
<td></td>
<td>$(\Phi^\top_i \Sigma^\top \omega) = \text{compTrace}(\Phi^\top_i \Sigma^\top_j \tau_m \xrightarrow{\psi} \tau_n \rightarrow \omega_f ; \Phi^\top_e \Sigma^\top_e \tau_j \xrightarrow{\psi} \tau_k)$</td>
</tr>
<tr>
<td></td>
<td>$\Delta; \Gamma; \Phi, \Sigma^\top \vdash e_f \Leftrightarrow e : \omega \rightarrow \Phi^\top_i \Sigma^\top$</td>
</tr>
</tbody>
</table>

Figure 5.9: Typing rules for strategy combinators
combinator environment, and sets the output strategy combinator environment as a singleton only containing \( x \) due to linearity.

**Typing Rule for Strategy Combinator Application**  The \( T\text{-S}\text{-App} \) rule is one of the most complex typing rules in this type system. It is for strategy combinator applications of the form \( e_f \leftarrow e \), that applies the strategy \( e \) to the strategy combinator \( e_f \). The rule calls \( \text{compTrace} \) computing the traced return type and environments by enumerating all possible ways of connecting each pair of traces in the type expected as a parameter by \( e_f \) and the type of \( e \) passed as an argument. It then applies the computation result to the type of \( e_f \)'s body and the types in the strategy combinator environment. Formally, \( T\text{-S}\text{-App} \) states that for a strategy combinator application \( e_f \leftarrow e \) where \( e_f \) has type \( (\tau_m \xrightarrow{\varphi_m} \tau_n) \Rightarrow \omega_f \) and \( e \) has type \( \tau_j \xrightarrow{\varphi} \tau_k \), we start with checking if the underlying structural types of the parameter \( e_f \) and the argument \( e \) are the same, that is, checking the equality of their types after trace erasure. Then we use the \( \text{compTrace} \) function to compute the type and the output environments of \( e_f \leftarrow e \). This process is much more involved than type-checking ordinary function applications because of the existence of traces.

**Trace Computation**  To understand how \( \text{compTrace} \) works, we look at its definition in Listing 5.1, which takes seven inputs that can be separated into two groups: the first four arguments \((\Phi_m; \Sigma_m; \omega_m; \omega_c)\) are the output environments and traced types from the strategy combinator \( e_f \), where \( \omega_m \) is the parameter type and \( \omega_c \) the type of the strategy combinator body; the other three arguments \((\Phi_n; \Sigma_n; \omega_n)\) are the output environments and the traced type from the argument \( e \). Due to linearity, \( \omega_m \) cannot share any common trace identifiers with \( \omega_n \). The trace computation is performed on \( \omega_m \) and \( \omega_n \), and the computation result is then applied to \( \Sigma_m \), \( \Sigma_n \), and \( \omega_c \) to synthesize the resulting environments and traced type \( (\Phi_r; \Sigma_r; \omega_r) \). We use \((=)\) for mutable variable assignments and \((=)\) for constant initialization. When computing with traced types \( \omega \), we must handle them together with their tracing and strategy combinator environments \( \Phi \) and \( \Sigma \). Therefore, we call them together a traced triple \((\Phi; \Sigma; \omega)\), where \( \Sigma \) and \( \omega \) is well-traced in \( \Phi \).

The basic idea of \( \text{compTrace} \) is to enumerate all possible ways of connecting the traces in \( \omega_m \) and \( \omega_n \) and applying the result to \( \omega_c \) and the strategy combinator environments. We start in lines 2–3 by extracting the trace identifier sets \( \varphi_m \) and \( \varphi_n \) from the traced triples for \( \omega_m \) and \( \omega_n \) using the \text{traceIds} function which simply returns the
**Listing 5.1: Definition of compTrace**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{compTrace}(\Phi_m; \Sigma_m; \omega_m; \omega_c; \Phi_n; \Sigma_n; \omega_n) = )</td>
</tr>
<tr>
<td>2</td>
<td>( \varphi_m = \text{traceIds}(\Phi_m; \omega_m) )</td>
</tr>
<tr>
<td>3</td>
<td>( \varphi_n = \text{traceIds}(\Phi_n; \omega_n) )</td>
</tr>
<tr>
<td>4</td>
<td>(( \Phi_m^-; \Sigma_m^-; \omega_c^- )) = \text{Delete } \varphi_m \text{ in } (\Phi_m; \Sigma_m; \omega_c)</td>
</tr>
<tr>
<td>5</td>
<td>(( \Phi_n^-; \Sigma_n^-; \omega_n^- )) = \text{Delete } \varphi_n \text{ in } (\Phi_n; \Sigma_n; \omega_n)</td>
</tr>
<tr>
<td>6</td>
<td>(( \Phi_r; \Sigma_r; \omega_r )) := ((( \Phi_m^-; \Phi_n^- )), (( \Sigma_m^-; \Sigma_n^- )), ( \omega_c^- ))</td>
</tr>
<tr>
<td>7</td>
<td>( \text{for } \alpha_m \text{ in } \varphi_m )</td>
</tr>
<tr>
<td>8</td>
<td>( \text{for } \alpha_n \text{ in } \varphi_n )</td>
</tr>
<tr>
<td>9</td>
<td>(( \alpha_m \triangleright \phi_m; \Sigma_m'; \omega_m' )) = \text{Select } [\alpha_m] \text{ in } (\Phi_m; \Sigma_m; \omega_m)</td>
</tr>
<tr>
<td>10</td>
<td>(( \alpha_m \triangleright \phi_m; \cdot; \omega_c' )) = \text{Select } [\alpha_m] \text{ in } (\Phi_m; \cdot; \omega_c)</td>
</tr>
<tr>
<td>11</td>
<td>(( \alpha_m \triangleright \phi_n; \Sigma_n'; \omega_n' )) = (( \text{Select } [\alpha_n] \text{ in } (\Phi_n; \Sigma_n; \omega_n) ))[\alpha_n \mapsto \alpha_m]</td>
</tr>
<tr>
<td>12</td>
<td>( \overline{\Sigma} = \text{unifyTrace}(\alpha_m \triangleright [\phi_m, \phi_n]; \omega_m', \omega_n') )</td>
</tr>
<tr>
<td>13</td>
<td>( \text{if } \overline{\Sigma} \neq \text{failTrace} )</td>
</tr>
<tr>
<td>14</td>
<td>(( \Phi_i; \Sigma_i; \omega_i )) = \text{Fresh } (( \alpha_m \triangleright [\phi_m, \phi_n]; \Sigma_m', \Sigma_n')[\overline{\Sigma}]; \omega_c'[\overline{\Sigma}] ))</td>
</tr>
<tr>
<td>15</td>
<td>(( \Phi_r; \Sigma_r; \omega_r )) := (( \Phi_i; \Sigma_i; \omega_i )) to (( \Phi_r; \Sigma_r; \omega_r ))</td>
</tr>
<tr>
<td>16</td>
<td>( \text{return } (\Phi_r; \Sigma_r; \omega_r) )</td>
</tr>
</tbody>
</table>

trace identifier set \( \varphi \) from a strategy type \( \tau_p \xrightarrow{\varphi} \tau_c \) or a strategy result type \( \varphi \triangleright \tau \). The extracted trace identifier sets contain the traces that are involved in the subsequent computations. In lines 4–6, we remove the involved traces from the traced triple of \( \omega_c \) and the traced triple of \( \omega_n \) to obtain traced triples that only contain traces that are not involved in the computation. We combine the triples into \( (\Phi_r; \Sigma_r; \omega_r) \), that is also the initial value of the accumulator for the subsequent loops. The two nested loops in lines 7–8 enumerate all pairs of trace identifiers \( (\alpha_m, \alpha_n) \) from \( \varphi_m \) and \( \varphi_n \). In the loop body, we select the traced triples traced by \( \alpha_m \) and \( \alpha_n \) (lines 9–11) from the traced triples of the input types, to get single-traced slices. We rename \( \alpha_n \) to \( \alpha_m \), in line 11, to ensure that all traces are temporarily uniformly identified and ready for subsequent computations. The Select function, selects traces from the tracing environment and removes all not selected traces from the traced triple it returns. Its formal definition is in the appendix in Figure 5.20. Select is the dual of the Delete operator used before. In line 12, we unify the traced types \( \omega_m' \) and \( \omega_n' \). We will discuss trace unification in more depth below. If the unification succeeded (line 13) with a result \( \overline{\Sigma} \), there is a way to connect the traces identified by \( \alpha_m \) and \( \alpha_n \), meaning that there exists a possible execution path for the strategy. Subsequently, in line 14, we apply \( \overline{\Sigma} \) to a traced triple containing the strategy combinator environments \( (\Sigma_m', \Sigma_n') \) which can be safely put
unifyTrace : (Φ; t; t) → S
unifyTrace(α ⊢ ϕ; (\{a\}; \{\})) = failTrace
unifyTrace(α ⊢ ϕ; (\{a\}; \{\})) = failTrace
unifyTrace(α ⊢ [β, ϕ]; τ_{β}; τ_{n}) = if τ_{n} = τ then failTrace else τ_{β} ↦→ τ_{n}
unifyTrace(α ⊢ [β, ϕ]; τ_{m}; τ_{β}) = if τ_{m} = τ then failTrace else τ_{β} ↦→ τ_{m}
unifyTrace(Φ; ⟨ρ_{m}; ρ_{n}⟩) = unifyTrace(Φ; ρ_{m}; ρ_{n})
unifyTrace(Φ; (τ_{m}, τ_{n}); (τ_{j}, τ_{k})) = let S = unifyTrace(Φ; τ_{m}; τ_{j})
        in unifyTrace(Φ; τ_{n}[S]; τ_{k}[S]) ◦ S
unifyTrace(Φ; τ; τ) = (· ↦→ ·)
unifyTrace(Φ; (ℓ : τ_{m} | ρ_{m}); (ℓ : τ_{n} | ρ_{n})) = let S = unifyTrace(Φ; τ_{m}; τ_{n})
        in unifyTrace(Φ; ρ_{m}[S]; ρ_{n}[S]) ◦ S
unifyTrace(Φ; ρ; ρ) = (· ↦→ ·)
unifyTrace(Φ; [α] ▶ τ_{m}; [α] ▶ τ_{n}) = unifyTrace(Φ; τ_{m}; τ_{n})
unifyTrace(Φ; τ_{m} \rightarrow [\ell] τ_{j} [\ell] τ_{k}) = let S = unifyTrace(Φ; τ_{m}; τ_{j})
        in unifyTrace(Φ; τ_{n}[S]; τ_{k}[S]) ◦ S

Figure 5.10: Definition of trace unification

together because of linearity, and the single-traced return type of e f (ω_{f}). We Freshly rename the traced triples before Adding it to the accumulator in line 15 to enforce the distinction between trace identifiers. Finally, the accumulator is returned as the result of compTrace (line 16). The formal definitions of Fresh and Add can be found in the appendix in Figure 5.22 and Figure 5.23.

Trace Unification Like ordinary type unifications, for traced type ω_{m} and ω_{n}, trace unification produces a trace substitution S such that ω_{m}[S] = ω_{n}[S]. Trace substitutions, defined in Figure 5.11, are collections of mappings, but when a mapping in a ordinary substitution replaces a variable with a term, each mapping in a trace substitution replaces a traceable type traced by a trace member with another traceable type, that is τ_{β}[S] = S(τ_{β}) if τ_{β} ∈ dom(S). Actually, there is no need to check τ while looking up τ_{β} in dom(S) because β is already the unique key for the corresponding mapping (see Lemma 5.3.1). Furthermore, trace substitutions are always generated
by `unifyTrace` in this type system, and `unifyTrace` can only be performed on types whose underlying types are the same, so we have \( \tau = \text{erase}(\tau[\beta]) \). In other words, trace substitutions never affect the underlying types. Trace substitutions naturally adapt the operations for ordinary substitutions, but for a concise formalization, we extend trace substitutions with a bottom value `failTrace` to represent the failure of `unifyTrace`, and the compositions of any trace substitutions and `failTrace` will result in `failTrace`.

A definition of the function `unifyTrace` is shown in Figure 5.10. The definition is straightforward and very close to a standard type-unification definition. `failTrace` is returned when we try to unify a traced type with its untraced counterpart, otherwise a trace substitution is returned. It is worth noting that `unifyTrace` is not defined for and never applied to strategy combinator types. Lemma 5.3.2 shows that `unifyTrace` is a correctly defined trace unification.

**Lemma 5.3.1 (Unique Underlying Type).** If \( \Delta \vdash e_m : \omega + \Phi \), for each \( \alpha \triangleright \phi \in \Phi \), there is \( (\alpha \triangleright \phi; : \omega_m) = \text{Select } [\alpha] \text{ in } (\Phi; ; \omega) \), and for each \( \beta \in \phi \), it always traces an unique \( \tau \) in \( \omega_m \).

**Proof.** By induction on the typing derivation of \( \Delta \vdash e_m : \omega + \Phi \). \( \square \)

**Lemma 5.3.2 (Unification).** If `unifyTrace`(\( \Phi;\omega_1;\omega_2 \)) = \( \overline{\Sigma} \neq \text{failTrace} \), then \( \omega_1[\overline{\Sigma}] = \omega_2[\overline{\Sigma}] \).

**Proof.** By induction on \( \omega_1 \) and the definition of `unifyTrace` and Lemma 5.3.1. \( \square \)
Examples of Typing Strategies  Figure 5.12 shows a simplified typing derivation to demonstrate the typing of the sequential composition from Example 5.3. In this figure, we write $e_{mn}$ and $e_{vw}$ for the two composed strategies and use the simplified types for visual clarity.

To type the expression $\text{seq} \leftarrow e_{mn} \leftarrow e_{vw}$, we type $\text{seq} \leftarrow e_{mn}$ and $e_{vw}$ separately and then compute the resulting traced type based on their types. At the top of the figure, we type $\text{seq}$ and $e_{mn}$ and use their types as input to the $\text{compTrace}$ function. In this instance, there are two iterations as we connect one trace from $\text{seq}$ ($s$) with two traces from $e_{mn}$ ($a$ and $b$), and each iteration performs trace unification. As a result, two iterations successfully give two traced types, which are combined into the result type of the strategy application. For the second strategy application in the bottom half of the figure, we compute the traces of two strategies each with two traces, therefore, $\text{compTrace}$ performs 4 ($= 2 \times 2$) iterations enumerating all possible ways to connect a trace from the first type with one from the second. Here, for not all iterations the trace unification is successful, as not all traces connect. The two successfully unified traced types are combined, resulting in the final traced type of the overall strategy application.

Figure 5.13 shows a simplified typing derivation to demonstrate the typing of the choice composition from Example 5.6. In this figure, we write $\text{rule } m + n \rightarrow n + m$ as $e_p$, $\text{rule } m + n \rightarrow n * n$ as $e_m$, and $\text{rule } 1 * v \rightarrow v$ as $e_0$.

According to Example 5.6, the warning is triggered by the sequential composition
\[
\text{seq} : (s_0 \xrightarrow{[x]} s_1) \Rightarrow (s_1 \xrightarrow{[x]} s_2) \Rightarrow (s_0 \xrightarrow{[x]} s_2)
\]

1. Iteration \( (f_1 + f_0 \xrightarrow{[f]} f_2) \Rightarrow (f_0 + f_1 \xrightarrow{f} f_2) = \text{compTrace}(\ldots) \)

\[
\text{seq} \leftarrow e_p : (f_1 + f_0 \xrightarrow{[f]} f_2) \Rightarrow (f_0 + f_1 \xrightarrow{f} f_2)
\]

1. Iteration \( \text{failTrace} \)

\[
\vdash \text{choice} : (a_0 \xrightarrow{[a]} a_1) \Rightarrow (b_0 \xrightarrow{[b]} b_1) \Rightarrow (a_0 \mid b_0 \xrightarrow{[a,b]} a_1 \mid b_1)
\]

T-Seq

\[
\vdash e_p : p_0 + p_1 \xrightarrow{[p]} p_1 + p_0
\]

T-S-App

\[
\vdash e_m : m_0 \cdot m_1 \xrightarrow{[m]} m_1 \cdot m_0
\]

T-S-App

WARNING

No Iteration

\[
\vdash \text{choice} \leftarrow (\text{seq} \leftarrow e_p \leftarrow e_m) : (b_0 \xrightarrow{[b]} b_1) \Rightarrow (b_0 \xrightarrow{[b]} b_1)
\]

T-S-App

\[
\vdash 1 \cdot a_0 \xrightarrow{[a]} a_0 = \text{compTrace}(\ldots)
\]

T-S-App

\[
\vdash e_v : 1 \cdot v_0 \xrightarrow{[v]} v_0
\]

T-S-App

\[
\vdash \text{choice} \leftarrow (\text{seq} \leftarrow e_p \leftarrow e_m) \leftarrow e_v : 1 \cdot a_0 \xrightarrow{[a]} a_0
\]

Figure 5.13: Simplified typing derivation for Example 5.6

\textbf{seq} \leftarrow e_p \leftarrow e_m, \text{ and this is marked in the figure with an orange WARNING. } \text{The typing derivation above shows more details: mismatched patterns in the two rewrite rules lead to a failed trace unification result, and the type produced by } \text{compTrace} (which is also orange) \text{ is empty-traced, indicating that the whole strategy combinator application does not have any successful execution paths. Afterwards, this empty-traced sequential composition, although will never produce successful rewriting results, is used as one branch in the choice composition. Thus, the final traced type only contains the tracing information of } e_v.
\[ \Delta; \Gamma \vdash \text{fail} : \varphi \triangleright \tau \triangleright \varphi \triangleright [] \quad \text{T-Fail} \]

\[ \begin{array}{l}
\text{fresh } \alpha \quad \Delta; \Gamma; \alpha \triangleright [] : \star \vdash e : \tau_e \\
\text{erase}(\tau_a) = \text{erase}(\tau_e) \\
(\varphi_a \triangleright [] ; \varphi_a \triangleright \tau_a) \to (\alpha \triangleright [] ; \alpha \triangleright \tau_e) \\
\end{array} \quad \Delta; \Gamma \vdash \text{succ} \ e : \varphi_s \triangleright \tau_s \triangleright \varphi_s \triangleright [] \\
\text{T-Succ} \]

\[ \begin{array}{l}
\Delta; \Gamma \vdash e_m : \varphi_m \triangleright \tau_m \triangleright \varphi_m \triangleright [] \\
\Delta; \Gamma \vdash e_n : \varphi_n \triangleright \tau_n \triangleright \varphi_n \triangleright [] \\
\text{erase}(\tau_m) = \text{erase}(\tau_n) \\
(\varphi_r \triangleright [] ; \varphi_r \triangleright \tau_r) = \text{Add} \ (\varphi_m \triangleright [] ; \varphi_m \triangleright \tau_m) \to (\varphi_n \triangleright [] ; \varphi_n \triangleright \tau_n) \\
\end{array} \quad \Delta; \Gamma \vdash e_m \boxplus e_n : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright [] \\
\text{T-Alt} \]

\[ \begin{array}{l}
\text{fresh } \alpha, \beta_m, \beta_n \\
\tau^f_m = \text{erase}(\tau_m)_{[\beta_m]} \\
\tau^f_n = \text{erase}(\tau_n)_{[\beta_n]} \\
(\Phi_i; \varphi_i \triangleright (\tau^f_m, \tau^f_n)) = \\
\text{compTrace}(\alpha \triangleright [\beta_m; \beta_n]; \alpha \triangleright \tau^f_m; \alpha \triangleright (\tau^f_m, \tau^f_n); \varphi_m \triangleright [] ; \varphi_m \triangleright \tau_m) \\
\text{erase}(\tau_p) = \text{erase}((\tau_m, \tau_n)) \\
(\varphi_r \triangleright [] ; \varphi_r \triangleright (\tau^f_m, \tau^f_n)) = \text{Add} \ (\varphi_p \triangleright [] ; \varphi_p \triangleright \tau_p) \to \\
\text{compTrace}(\Phi_i; \varphi_i \triangleright \tau^f_n; \varphi_i \triangleright (\tau^f_m, \tau^f_n); \varphi_m \triangleright [] ; \varphi_m \triangleright \tau_m) \\
\end{array} \quad \Delta; \Gamma \vdash e_m \Box e_n : \varphi_r \triangleright (\tau^f_m, \tau^f_n) \triangleright \varphi_r \triangleright [] \quad \text{T-Pair} \]

\[ \begin{array}{l}
\Delta; \Gamma \vdash e_f : \tau_p \xrightarrow{\varphi_f} \tau_e + \Phi_f \\
\Delta; \Gamma \vdash e_i : \varphi_i \triangleright \tau_i \triangleright \varphi_i \triangleright [] \\
\text{erase}(\tau_p) = \text{erase}(\tau_i) \\
(\varphi_r \triangleright [] ; \varphi_r \triangleright \tau_r) = \text{compTrace}(\Phi_f; \varphi_f \triangleright \tau_p; \varphi_f \triangleright \tau_e; \varphi_i \triangleright [] ; \varphi_i \triangleright \tau_i) \\
\end{array} \quad \Delta; \Gamma \vdash e_f \leftarrow e_i : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright [] \quad \text{T-Exec} \]

Figure 5.14: Typing rules for strategy execution and results
5.3.6 Typing Rules for Strategy Executions and let-bindings

Typing Rules for Strategy Execution  Being able to define rewrite rules and compose them with strategy combinators, we still need to execute the strategies and get the results. Figure 5.14 shows the typing rules for strategy execution and results. Executing a strategy gives two possible outcomes, \texttt{fail} or \texttt{succ }\( e \), and the execution is expected to preserve types. Thus, the typing rule \texttt{T-Fail} allows the \texttt{fail} result to have an arbitrary underlying type and arbitrary traces because we know nothing about the strategy execution returning \texttt{fail}. It is worth noting that the output tracing environment does not contain any trace members because there cannot be any pattern variables in a closed term. Actually, all the expressions of the strategy result type are closed with respect to the rule environments and strategy combinator environments because in practice we always apply strategies to ASTs which are closed terms in the strategy language. In the rule \texttt{T-Succ}, if \texttt{succ }\( e \) is the execution result of a strategy, the expression \( e \) must have been constructed by the RHS of a rewrite, so there must be a trace for \( e \) and \( e \) is typed in the same way as the RHS of a rewrite rule. Similarly, we also allow arbitrary traces to be added for type preservation. The \texttt{T-Alt} rule is for the non-deterministic choice operator \( \sqcup \) which choose one from two operands. It is intuitive to require the two operands to have the same underlying type and Add their traces together. The \texttt{T-Pair} rule is for the pairing operator \( \checkmark \) which creates a pair and puts two operands into it. This typing rules looks complicated, but it is just enumerating all possible ways of pairing the two results, since each execution result can have multiple traces and each trace represents a possible rewrite outcome. It creates a freshly single-traced pair \( [\alpha] \triangleright (\tau_m^f,\tau_n^f) \) to hold the intermediate computation result. In the first \texttt{compTrace}, the pair replicates the traces in \( \varphi_m \triangleright \tau_m \), and in the second \texttt{compTrace} it finishes the enumeration by computing all possible connections with the traces in \( \varphi_n \triangleright \tau_n \). Finally, the relatively complex rule \texttt{T-Exec} is for strategy executions. The structure and mechanism of this typing rule is almost the same as those of the \texttt{T-S-App} rule for strategy combinator applications, and the \texttt{compTrace} function is also used to compute the traces in the execution results.

Typing Rules for Let-Bindings  The last typing rules are shown in Figure 5.15, including the rules for \texttt{let}-binding, the usage of \texttt{let}-bound variables, and the type variable instantiation rules. All these rules are straightforward. Considering that the abstractions of traces and the underlying type variables are syntactically separated, the operations on the underlying types in these typing rules are also standard. Since
our goal is to let traces precisely reflect the internal rewriting paths of strategies, the abstractions and instantiations of traces should preserve as much tracing information as possible while avoiding violating the sub-structural rules of the tracing environments. Thus, in the T-Let rule, the output tracing environment $\Phi_f$ of $e_f$ is directly bound by the tr keyword in the type scheme, and in the T-Var rule, the bound tracing environment is freshly renamed (together with the traces in the traced type). By alpha-equivalence, the tracing information does not change, and there will not be any name collisions because all trace variables are freshly renamed. Besides, the T-Let rule will trigger the error in Example 5.7 and Example 5.8 by requiring that $\Phi_f$ is non-empty. The type variable instantiation rules are of the form $\Delta \vdash \sigma \leq \omega$, stating that the $\sigma$ is a type scheme without a tracing environment, and it can be instantiated to a traced type $\omega$ in the kinding environment $\Delta$. The instantiations for rows and types are standard, except that the types or rows the variables are instantiated to must not contain any trace variables, so we can clearly distinguish the trace and type variable abstractions and preserve the tracing information.

**Example of Let-Bindings** Figure 5.16 shows a simplified typing derivation to demonstrate the typing of the let-binding from Example 5.7. In this figure, we write rule 2 * n -> n + n as $e_n$ and rule 2 + 3 -> 5 as $e$. The type error is marked in the figure with a red ERROR. The typing of the body of the let-binding, a sequen-
tial composition, is straightforward, and an empty-traced type (which is also red) is produced. The T-Let rule checks the tracing information of the body of this let-binding, and treats the empty-traced body as a type error because no matter where the let-bound variable $x$ is used, it is guaranteed to fail at runtime.

**Workaround for Linearity of Strategy Variables**  Recalling the linear restriction mentioned in Section 5.3.2, we forbid the definition of strategy combinators containing repeated usage of strategy variables such as $\text{st } x \Rightarrow \text{seq } \Leftarrow x \Leftarrow x$. A workaround for this restriction is that we can define the strategy in a let-binding and repeatedly use the polymorphic let-bound variable, $f$ for example, because for each time $f$ is used, the bound tracing environment will be freshly renamed and there will not be common trace variables in the types of all occurrences of $f$.

5.3.7 Operational Semantics

Figure 5.17 shows the syntax of values (denoted as $v$) and evaluation contexts (denoted as $E$) defined in the style used in [Wright and Felleisen, 1994]. In Core $S_t$, primitive strategy combinators and their applications to values are considered as a subset of values, the combinator values (denoted as $cv$). From this perspective, primitive strategy combinators are similar to the data constructors in Haskell. The rest of the value and evaluation context definitions are straightforward.

Figure 5.18 shows the reduction rules in the operational semantics of Core $S_t$ and the definition of a function $\text{match}$ performing pattern matching. The relation $e_m \rightsquigarrow e_n$
Combinator Values \( cv := \text{seq} | \text{choice} | \text{both} | cv \leftarrow v \)

Values \( v := cv | \text{st} x \Rightarrow e_b | \text{rule} p \rightarrow e | \ell v | (v_m, v_n) | \text{succ} v | \text{fail} \)

Evaluation Contexts \( E := [] | E \leftarrow e | v \leftarrow E | E \leftarrow E | \ell E | (E, e) | (v, E) | \text{let} x = E \text{ in } e | \text{succ} E | E \boxplus e | v \boxplus E | E \boxplus e | v \boxplus E \)

Figure 5.17: Syntax of values and evaluation contexts.

contains the notions of reduction which are the interesting cases of \( e_m \) reducing to \( e_n \), and by embedding this relation into the evaluation contexts, we get the stepping relation \( e_m \rightarrow e_n \) which works for any redex Wright and Felleisen [1994]. All these definitions are straightforward. Being similar to \( \text{unifyTrace} \), the substitutions produced by \( \text{match} \) are also extended with a bottom value \( \text{failPat} \) to represent failed pattern matching. Lemma 5.3.3 shows that the \( \text{match} \) function is correctly defined.

**Lemma 5.3.3 (Pattern Matching).** If \( \text{match}(p; v) = S \neq \text{failPat} \), then \( p2e(p)[S] = v \).

**Proof.** By the definition of \( \text{match} \) and induction on \( p \). \( \square \)

It is worth noting that the introduction of combinator values is a design choice we have made. As shown by the reduction rules for \text{seq}, \text{choice} and \text{both}, the fully applied primitive strategy combinators are interpreted as the compositions of strategy executions. Apparently, if we have the lambda abstraction of the execution results, it is possible to define these primitive combinators inside the language instead of introducing them as primitive constructs. We intentionally avoided the lambda abstraction of the execution results because otherwise the definition of rewrite rules can be very complicated. Another benefit of this design choice is that it also makes the extension of primitive strategy combinators easier.

### 5.3.8 Soundness and Strong Normalization

In this section, we show the soundness of our tracing type system, and its strong normalization property. In this and the subsequent sections, for a relation \( R \), we write \( R^+ \) for its transitive closure, and \( R^* \) for its reflexive transitive closure.
\[ E[e_a] \leadsto E[e_b] \text{ iff } e_a \bowtie e_b \]

\[ v \leftarrow \text{fail} \bowtie \text{fail} \]

\[ \text{(seq) } \leftarrow v_a \leftarrow v_b \leftarrow v_i \leadsto (v_b \leftarrow v_a \leftarrow v_i) \]

\[ \text{(choice) } \leftarrow v_a \leftarrow v_b \leftarrow v_i \leadsto (v_a \leftarrow v_b \leftarrow v_i) \]

\[ \text{(both) } \leftarrow \text{succ}(v_m, v_n) \leadsto (v_a \leftarrow \text{succ} v_m)(v_b \leftarrow \text{succ} v_n) \]

\[ \text{(rule } p \rightarrow e \text{) } \leftarrow \text{succ } v \leadsto \text{fail} \text{ if match}(p; v) = \text{failPat} \]

\[ \text{(rule } p \rightarrow e \text{) } \leftarrow \text{succ } v \leadsto \text{succ } e[S] \text{ if match}(p; v) = S \neq \text{failPat} \]

\[ \text{fail} \bowtie \text{succ } v_b \leadsto \text{succ } v_b \]

\[ \text{succ } v_a \bowtie \text{succ } v_b \leadsto \text{succ } v_a \]

\[ \text{succ } v_a \bowtie \text{succ } v_b \leadsto \text{succ } v_b \]

\[ \text{succ } v_a \bowtie \text{fail} \leadsto \text{succ } v_a \]

\[ \text{fail} \bowtie \text{fail} \leadsto \text{fail} \]

\[ \text{fail} \bowtie \text{succ } v_b \leadsto \text{fail} \]

\[ \text{succ } v_a \bowtie \text{succ } v_b \leadsto \text{succ } (v_a, v_b) \]

\[ \text{succ } v_a \bowtie \text{fail} \leadsto \text{fail} \]

\[ \text{fail} \bowtie \text{fail} \leadsto \text{fail} \]

\[ (\text{st } x \Rightarrow e) \leftarrow v \leadsto e[x \mapsto v] \]

\[ \text{let } x = v \text{ in } e \leadsto e[x \mapsto v] \]

\[ \text{match} : p \rightarrow v \rightarrow S \]

\[ \text{match}(x; v) = x \mapsto v \]

\[ \text{match}(\ell p; \ell v) = \text{match}(p; v) \]

\[ \text{match}(();()) = (\cdot \mapsto \cdot) \]

\[ \text{match}((p_a, p_b);(v_a, v_b)) = \text{match}(p_b; v_b) \circ \text{match}(p_a; v_a) \]

\[ \text{match}(p; v) = \text{failPat} \]

Figure 5.18: Operational Semantics
Before presenting the type soundness theorem, we show below a series of substitution lemmata. Lemma 5.3.4 and Lemma 5.3.5 are for the executions of rewrite rules. Lemma 5.3.4 states that if the pattern matching in a non-empty traced rewrite rule execution succeeds, the execution is type-preserving, while Lemma 5.3.5 states that an empty-traced rewrite rule execution must fail.

**Lemma 5.3.4 (Rule Substitution).** If $\Delta; \Gamma \vdash (\text{rule } p \rightarrow e) \leftarrow \text{succ } v : [\alpha_r] \triangleright \tau_r \triangleright \alpha_r \triangleright []$, and $\text{match}(p; v) = S \neq \text{failPat}$, then $\Delta; \Gamma \vdash \text{succ } e[S] : [\alpha_r] \triangleright \tau_r \triangleright \alpha_r \triangleright []$.

*Proof.* The proof proceeds by induction on the derivation of the typing judgement for $p$. See full proof on page 154. □

**Lemma 5.3.5 (Failed Rule).** If $\Delta; \Gamma \vdash (\text{rule } p \rightarrow e) \leftarrow \text{succ } v : [] \triangleright \tau_r$, then $(\text{rule } p \rightarrow e) \leftarrow \text{succ } v \leadsto \text{fail}$.

*Proof.* The proof proceeds by induction on the derivation of the typing judgement for $p$. See full proof on page 155. □

Lemma 5.3.6 shows that the beta-reductions of strategy combinator applications are type-preserving. It also justifies the design of the typing rule T-S-Lam.

**Lemma 5.3.6 (Strategy Substitution).** If $\Delta; \Gamma; \Phi \downarrow; \Sigma \downarrow \vdash (\text{st } x \Rightarrow e_f) \Leftarrow e : \omega \rightarrow \Phi \uparrow; \Sigma \uparrow$, then $\Delta; \Gamma; \Phi \downarrow; \Sigma \downarrow \vdash e_f[x \mapsto e] : \omega \rightarrow \Phi \uparrow; \Sigma \uparrow$.

*Proof.* The proof proceeds by induction on the derivation of the typing judgement for $e_f$. See full proof on page 155. □

With the lemmata shown here and with more detail in the appendix, the proofs for the two parts of type soundness, preservation (Corollary 5.3.7.1) and progress (Theorem 5.3.8), are straightforward.

**Theorem 5.3.7 (Subject Reduction).** If $\Delta \vdash e_m : \omega \rightarrow \Phi$, and $e_m \leadsto e_n$, then $\Delta \vdash e_n : \omega \rightarrow \Phi$.

*Proof.* By case analysis on the reduction $e_m \leadsto e_n$. Proof for cases that cannot be routinely proven are provided by Lemma 5.3.4, Lemma 5.3.6 and Lemma 5.B.1. □

**Corollary 5.3.7.1 (Preservation).** If $\Delta \vdash e_m : \omega \rightarrow \Phi$, and $e_m \rightarrow e_n$, then $\Delta \vdash e_n : \omega \rightarrow \Phi$.

*Proof.* This is a direct corollary of Theorem 5.3.7 by the definition of $\rightarrow$. □

**Theorem 5.3.8 (Progress).** If $\Delta \vdash e_m : \omega \rightarrow \Phi$, then either $e_m$ is a value, or there exists an $e_n$ such that $e_m \rightarrow e_n$. 


Proof. We need to show that if $\Delta \vdash e_m : \omega \rightarrow \Phi$, then either $e_m$ is a value, or there exists an evaluation context $E$ and two expressions $e_j$ and $e_k$ such that $e_m = E[e_j]$ and $e_j \leadsto e_k$. The proof routinely proceeds by induction on the typing derivation of $\Delta \vdash e_m : \omega \rightarrow \Phi$.

Corollary 5.3.8.1 (Type Soundness). If $\Delta \vdash e : \omega \rightarrow \Phi$, then either $e$ is a value, or there exists an $e_n$ such that $e_m \rightarrow^+ e_n$ and $\Delta \vdash e_n : \omega \rightarrow \Phi$.

Proof. This is a direct corollary of Corollary 5.3.7.1 and Theorem 5.3.8.

Besides type soundness, Core $S_t$ also has the strong normalization property. The proof technique used in the strong normalization proof of STLC Pierce [2002] can also be applied here.

Theorem 5.3.9 (Strong Normalization). If $\Delta \vdash e : \omega \rightarrow \Phi$, then $e \rightarrow^* v$ where $v$ is a value.

Proof. It is easy to show that $\Delta; \Gamma \vdash (\text{rule } p \rightarrow e) \leftarrow e_j : \varphi_r \triangleright \tau_r \leftarrow \Phi_r$ always normalize, so there is no need to consider substitution for variables from $\Theta$. Thus, for a term $e$, we need to show that after substituting strong-normalizing terms for variables from $\Gamma$ (by $S_\Gamma$) and $\Sigma$ (by $S_\Sigma$) in $e$, $e[S_\Gamma][S_\Sigma]$ is also strong-normalizing. This proof proceeds by constructing the logical relation on strong normalizing terms indexed by type $\omega$ and then induction on $e$.

5.3.9 Properties of Well-typed Strategies

This section gives formal statements about the properties of well-typed strategies. Corollary 5.3.9.1 and Corollary 5.3.9.2 are about well-typed but empty-traced expressions. Corollary 5.3.9.1 shows that an expression of an empty-traced result type must evaluate to $\text{fail}$. Corollary 5.3.9.2 refines the condition in Corollary 5.3.9.1 and states that if a strategy is empty-traced, then for any well-typed execution of this strategy, the result will be $\text{fail}$. The emptiness check for the tracing environment in $T$-Let is supported by Corollary 5.3.9.2 because a strategy known to always fail is useless or unproductive.

Corollary 5.3.9.1 (Empty-traced Failed Result). If $\Delta \vdash [\textbf{}] \triangleright \tau$, then $e \rightarrow^* \text{fail}$.

Proof. This is a direct corollary of Theorem 5.3.9 and Corollary 5.3.8.1 because $\text{fail}$ is the only value can be typed with $[\textbf{}] \triangleright \tau$. 

□
Corollary 5.3.9.2 (Empty-traced Unproductive Strategy). If $\Delta \vdash e_f : \tau_a \rightarrow \tau_b$, then for any $e_i$ that $\Delta \vdash e_f \leftarrow e_i : [] \triangleright \tau_r$, there is $(e_f \leftarrow e_i) \rightarrow^+ \text{fail}$.

Proof. This is a direct corollary of Corollary 5.3.9.1 and T-Exec. $\square$

As mentioned in Section 5.3.6, for type preservation, arbitrary traces can be added during the typing of $\textsf{succ} e$. This allows more traces to be computed for the strategy execution using $\textsf{succ} e$ as its input, making the tracing result less precise. To formally and precisely define the meaning of “well-traced”, we introduce the concept of the minimal trace identifier set.

Definition 5.3.1 (Minimal Trace Identifier Set). With the conditions specified below, we say an expression ($e$) of the strategy result type ($\varphi \triangleright \tau$) is typed with a minimal trace identifier set ($\varphi$).

- If $\Delta; \Gamma \vdash \text{fail} : [] \triangleright \tau$, then $\text{fail}$ is typed with a minimal trace identifier set $[]$.
- If $\Delta; \Gamma \vdash \text{succ} e : [\alpha] \triangleright \tau_s \uplus \alpha \triangleright []$, then $\text{succ} e$ is typed with a minimal trace identifier set $[\alpha]$.
- If $e_m$ and $e_n$ are typed with minimal trace identifier sets in the derivation of $\Delta; \Gamma \vdash e_m \boxplus e_n : \varphi_r \triangleright \tau_r \uplus \varphi_r \triangleright []$, then $e_m \boxplus e_n$ is typed with a minimal trace identifier set $\varphi_r$.
- If $e_m$ and $e_n$ are typed with minimal trace identifier sets, and $\varphi_p = []$ (see T-Pair) in the derivation of $\Delta; \Gamma \vdash e_m \boxchar e_n : \varphi_r \triangleright (\tau'_m, \tau'_n) \uplus \varphi_r \triangleright []$, then $e_m \boxchar e_n$ is typed with a minimal trace identifier set $\varphi_r$.
- If $e_i$ is typed with a minimal trace identifier set in the derivation of $\Delta; \Gamma \vdash e_f \leftarrow e_i : \varphi_r \triangleright \tau_r \uplus \varphi_r \triangleright []$, then $e_f \leftarrow e_i$ is typed with a minimal trace identifier set $\varphi_r$.

The definition of the minimal trace identifier set covers all ways in which expressions of the strategy result type can be constructed. In other words, the minimal trace identifier set is defined for any expression of the strategy result type. It is also straightforward to inductively show that the minimal trace identifier set is actually the smallest possible trace identifier set we can construct for the type of the corresponding expression, and is unique up to renaming of variables. By excluding all unnecessary traces, the type of the whole expression is precise.

With the minimal trace identifier set, we are able to define the meaning of well-traced strategy execution as in Definition 5.3.2.
Definition 5.3.2 (Well-traced Strategy Execution). If a well-typed strategy execution is typed with a non-empty minimal trace identifier set, then this strategy execution is well-traced, and its type can never be empty-traced.

Lemma 5.3.10 (Successful Rule). If for $e = (\text{rule } p \rightarrow e) \leftarrow \text{succ } v$, there is $\Delta; \Gamma \vdash e : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright \emptyset$ where the minimal $\varphi_r \neq \emptyset$, then $e \rightarrow^+ \text{succ } v_s$.

Proof. The proof proceeds by induction on the derivation of the typing judgement for $p$. See full proof on page 160.

As the generalization of Lemma 5.3.10 to strategies, Lemma 5.3.11 shows each trace corresponds to a possible execution path in a strategy execution.

Lemma 5.3.11 (Enumeration). If $\Delta; \Gamma \vdash v_f \leftarrow \text{succ } v_i : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright \emptyset$ where the minimal $\varphi_r \neq \emptyset$, then for each $\alpha_r$ in $\varphi_r$, there exists a reduction $(v_f \leftarrow \text{succ } v_i) \rightarrow^+ \text{succ } v_r$.

Proof. The proof proceeds by induction on $v_f$. See full proof on page 160.

Finally, Corollary 5.3.11.1 shows that well-typed and well-traced strategy executions must succeed. More precisely, this should be stated as "there must exist at least one successful execution result" because of non-determinism.

Corollary 5.3.11.1 (Successful Rewrite). If $\Delta \vdash e_f \leftarrow e_i : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright \emptyset$ where the minimal $\varphi_r \neq \emptyset$, then there exists a reduction $(e_f \leftarrow e_i) \rightarrow^+ \text{succ } v_s$.

Proof. This is a direct corollary of Lemma 5.3.11 and Corollary 5.3.8.1.

5.4 Discussion

As a strategy language, Core $S_t$ has shown its ability to properly express the basic component of strategic rewriting, rewrite rule, where both the LHS and the RHS are expressed as the combination of labels, pairs, and units. It also provides basic strategy combinators such as seq, choice, and both, for the sequential composition, non-deterministic choice, and paired execution of strategies, respectively. Furthermore, Core $S_t$ allows the definition of customized strategy combinators using the st keyword. In
combination with the \texttt{let}-binding, the users can assign names to rewrite rules and customized strategy combinators, and use them in different contexts polymorphically.

The novel feature in Core \( S_t \) is its traced structural type system. Compared with other strategy languages which are untyped (Stratego, for example), or are equipped with many-sorted type systems classifying terms into different nominal sorts (\( S'_\gamma \) [Lämmel, 2003], for example), Core \( S_t \) describes the shape of the transformed terms with structural types and includes a tracing system encoding the successful execution paths into types. With this traced type system, Core \( S_t \) can statically detect strategy compositions which are guaranteed to fail at runtime. Compared with other works for a similar purpose [Mametjanov, 2010; Mametjanov et al., 2011], Core \( S_t \) provides more compact types and a more general, also more systematic solution.

As summarized by Kirchner [2015], there are some commonly agreed language features which are expected to be present in a strategy language. In the list below, we evaluate Core \( S_t \) according to these language features (marked in \textbf{bold}).

- The definition of \textbf{elementary strategies}, or rewrite rules, is done by the \texttt{rule} keyword in Core \( S_t \). As mentioned in Section 5.5, it is also possible to extend Core \( S_t \) with rewrite conditions.

- \textbf{Building derivations} in Core \( S_t \) is implemented by the \texttt{seq} combinator which sequentially compose two strategies.

- \textbf{Selection of branches} in Core \( S_t \) is implemented by the \texttt{choice} combinator which makes nondeterministic choice between two branches. Section 5.4.2 explains the reason of preferring nondeterministic choice and how the properties of Core \( S_t \) will be affected if left-biased choice combinator is used.

- \textbf{Conditional}, or in other words, the \texttt{if} \texttt{cs} \texttt{then} \texttt{ts} \texttt{else} \texttt{es} construct, is not directly provided in Core \( S_t \) as a primitive combinator, but it can be implemented as a customized combinator without introducing any new language features. More detailed explanation is in Section 5.5.3.

- \textbf{Recursive strategies} as an important feature in many strategy languages, unfortunately, is not supported in Core \( S_t \) currently because of the conflict between recursive definitions and precise tracing. More detailed discussion is in Section 5.4.2.

- \textbf{Traversal combinators} are considered as a core feature in many strategy languages because it is a common need to investigate or traverse the structure of
Chapter 5. Traced Types for Safe Strategic Rewriting

the rewritten terms. As shown in Section 5.2, Core $S_t$ has the ability to define one-level traversal combinators which decompose the terms until reach the target position, apply the strategies, and then reconstruct the terms. However, without recursive definitions, Core $S_t$ does not support recursive traversals such as $\text{bottomUp}$. As mentioned in Section 2.1, many strategy languages provide generic traversal combinators which can be used on arbitrary AST node without defining the specific traversal process. Fully-fledged generic traversal is not currently supported in Core $S_t$ due to the lack of recursive definitions, but it is possible to add a limited form of generic one-level traversal to Core $S_t$ as an extension. More detailed discussion is in Section 5.4.2.

5.4.1 Comparison With Rewrite $S_t$

In this section, we show that Core $S_t$ has addressed most of the issues of Rewrite $S_t$ discussed in Section 4.6, and provides a type system with stronger formal guarantees, simplified formalization, and more flexibility.

Stronger Formal Guarantees

Core $S_t$ is able to provide stronger formal guarantees compared with Rewrite $S_t$. By removing the rewrite condition from rewrite rule definition, the success or failure of each individual rewrite rule is only decided by pattern matching, whose behaviours can be precisely reflected by traces at the type-level. By using a non-deterministic choice combinator, whose comparison with its deterministic counterpart will be discussed later in Section 5.4.2, traces have better correspondence with the behaviours of the choice composition. By switching to an operational semantics defined directly on the term syntax, both types and traces are preserved during the evaluation of terms, and formal statements such as the type soundness theorem are proved in the presence of traces. Therefore, while Rewrite $S_t$ can only talk about the properties of well-typed but empty-traced strategy executions, Core $S_t$ formally defines the meaning of "well-traced" and by Corollary 5.3.11.1 shows that a well-typed and well-traced strategy "can’t go wrong", as its execution is guaranteed to have a possible successful execution path.
Simplified Formalization

The formalization of Core $S_t$ is significantly simplified compared with Rewrite $S_t$. The tracing mechanism in Core $S_t$ directly traces types instead of tracing labels and type variables like Rewrite $S_t$. This tracing mechanism requires less trace variables, especially when variant types based on rows are the major tracing targets, and it no longer needs some auxiliary constructs like type boards. Importantly, it does not compromise on precision. In Core $S_t$, row-based extensible records are replaced by simple pairs because unlike row-based variants where the terminal row variables are not traced, the terminal row variables in records need to be traced to ensure the precision of traces, resulting in complex tracing mechanism as in Rewrite $S_t$. Although this change makes the encoding of ASTs in Core $S_t$ less flexible compared with Typed EL-EVATE and Rewrite $S_t$, the significantly simplified formalization and other improvements overshadow this minor loss of flexibility. Removing the rewrite condition and abandoning translation-based operational semantics are also helpful for simplification because there is no need to include an untraced sub-language as part of the formalization. Besides, the trace computations in Core $S_t$ are defined in an algorithmic style, reducing the extensive usage of substitutions as in Rewrite $S_t$.

More Flexibility

With the improved tracing mechanism, equi-recursive types are supported in Core $S_t$, allowing more flexible strategy compositions. Section 5.2 shows Core $S_t$ examples containing recursive types which cannot be traced in Rewrite $S_t$, and in Section 5.3.2, we show the folding / unfolding conversion rule of equi-recursive variant types in the presence of traces, which is only possible if types (instead of labels) are directly traced. However, as explained in Section 5.3.2, the non-linear usage of strategy variables and recursive strategy combinator definitions stay challenging for Core $S_t$. In the next section, we will discuss a possible approach to limited forms of recursion.

5.4.2 Design Choices and Trade-Offs

As explained in Chapter 1, the language design in this thesis is based on three assumptions, and one of them is "Tracing information needs to be completely precise". We have made some central design choices for Core $S_t$ to prioritize strong formal guarantees and precise types / traces in our type system. As consequences, of our design choices we do not (yet) support recursive strategies, generic traversals, and we
only support a non-deterministic version of choice. Strategy languages such as EL-EVATE and Stratego support recursion, generic traversals, and offer a deterministic version, called left-choice.

In this section, we discuss trade-offs in the design of these features and possible ways to support them without weakening our formal results too much and losing our precise types / traces completely.

**Deterministic Left Choice**

Core $S_t$ only supports a non-deterministic choice combinator, whereas Stratego introduced a deterministic left-choice combinator that first applies the strategy on the left, and only attempts to apply the strategy on the right if the first fails. To understand our design choice, we look at the strategy $e_8$, that is defined as a choice composition sequentially followed by another rewrite rule:

```plaintext
let e8 = (rule m * n -> n * m || rule 1 * v -> v) ;
    rule m + n -> n + m
```

The choice composition contains two rules for multiplications, and has the following type:

```
(rule m * n -> n * m ||
rule 1 * v -> v) : (a_0 * a_1) | (1 * b_0) \[a,b\] \(\longrightarrow\) (a_1 * a_0) | b_0
```

The rewrite rule swapping the operands of an addition has the following type:

```
rule m + n -> n + m : c_0 + c_1 \[c\] \(\longrightarrow\) c_1 + c_0
```

Thus, with the rules of our type system their sequential composition has the following type:

```plaintext
let e8 = (rule m * n -> n * m || rule 1 * v -> v) ;
    rule m + n -> n + m :
        1 * (d_0 + d_1) \[d\] \(\longrightarrow\) d_1 + d_0
```

There is only one trace identified by $d$ in the final strategy type, reflecting that the strategy expects inputs in the form of $1 * (d_0 + d_1)$. With such inputs the Corollary 5.3.11.1 guarantees, that the overall strategy execution must have one successful execution path, which is $rule 1 * v -> v$ followed by $rule m + n -> n + m$. However, maybe surprisingly, this would not be true when we use the left-biased deterministic choice combinator. This strategy execution would always return $fail$. The left-biased choice combinator starts with applying the first strategy to the input,
as the pattern \( m \times n \) overlaps with \( 1 \times v \), the right strategy is unreachable at runtime and the left strategy will always be chosen. But, unfortunately, the left strategy does not compose sequentially with the additional rewrite rules, so the overall strategy execution always fails.

If we want to make similar formal statements about the properties of well-typed strategies as in Section 5.3.9, but with left-biased choice, we will find that Corollary 5.3.9.1 and Corollary 5.3.9.2 still hold because they only rely on type soundness and strong normalization, which are not affected by switching to left-choice. On the other hand, Corollary 5.3.11.1 will not hold any more because as shown by the example above, with left-choice, the traces lose their correspondence with the execution paths. In other words, we will not be able to prove Lemma 5.3.11 with left-choice. To obtain a property stating that well-typed and well-traced strategy executions must succeed, we would need to statically remove the traces containing unreachable patterns from the types, making the design of a completely precise tracing mechanism much more challenging. This difficulty becomes more significant for customized strategy combinators containing left-choice. To precisely identify the traces of unreachable patterns, the type of each parameter strategy may need to carry the pattern information of all surrounding strategies, because the actual argument strategy can affect the reachability of the surrounding strategies. For example,

\[
\text{let } e_9 = \text{st } s \Rightarrow (s \mid \text{rule } 1 \times v \rightarrow v) \; ; \; \text{rule } m \times n \rightarrow n \times m : \quad
\begin{array}{c}
(a_0 \rightarrow \alpha + \beta) \Rightarrow (a_0 \mid 1 \times (b_0 + b_1) \rightarrow \alpha + \beta | \beta + b_0) \\
\end{array}
\]

\( e_9 \) is defined based on \( e_8 \) by abstracting over \( \text{rule } m \times n \rightarrow n \times m \), and the type of \( e_9 \) is assigned assuming non-deterministic choice is used. If we use left-biased choice instead and enforce precise tracing, according to the analysis of \( e_8 \) above, we have to remove all traces when \( \text{rule } m \times n \rightarrow n \times m \) is supplied as the argument of \( e_9 \), but we do not need to do so if \( \text{rule } 2 \times n \rightarrow n + n \) is the argument. In other words, we do not know if \( \text{rule } 1 \times v \rightarrow v \) is reachable until we see the actual argument and substitute it for \( s \). However, this substitution happens at runtime. To be able to check the reachability statically, the pattern \( 1 \times v \) needs to appear in the type of the parameter \( s \) and connect with trace \( b \), because if \( 1 \times v \) is unreachable, trace \( b \) needs to be removed.

These two examples only show a naive aspect of reachability check. The required changes to support precisely traced left-biased choice are currently unclear to us, but they will significantly complicate the design of the tracing mechanism, and make the
proof of formal guarantees more challenging. Thus, we made the design choice of only supporting non-deterministic choice combinator.

Recursion

As explained in Section 5.3.2, it is currently unclear to us how to precisely represent the traces for repeatedly used strategy variables, and similar limitation occurs for recursively defined strategy combinators because the traces attached to the combinator type will be used to compute itself. For example, assuming the `let`-binding supports recursive definition and the `||` combinator is left-biased, the repeat combinator can be defined as follows.

```plaintext
let repeat = st x => (x ; repeat x) || id
```

The repeat combinator repeatedly applies the argument strategy `x` until no applicable term exists. Since the unrolling of `repeat x` contains the sequential composition of `x` and itself, the input and output types of `x` are unified. To proceed with this typing attempt, although this definition is actually not typeable in Core $S_t$, we forcibly assign a single traced type $(\tau_0 \xrightarrow{\alpha_0} \tau_0) \Rightarrow (\tau_0 \xrightarrow{\alpha_0} \tau_0)$ to `repeat`, and $\tau_0 \xrightarrow{\alpha_0} \tau_0$ to `x ; repeat x`. Now we consider the choice composition with `id`, which adds another trace and makes the body of the definition `(x ; repeat x) || id` have type $\tau_1 \xrightarrow{[\alpha_0, \alpha_1]} \tau_1$. This result conflicts with the single traced type of `repeat`. Actually, no matter how we type `repeat`, this conflict always exists because the choice composition will always add one extra path and change the return type of `repeat`.

This example also refutes the possibility of tracing or approximating the traces of recursive definitions by unrolling. A possible workaround is extending Core $S_t$ with new base types (in a similar way to the unit type) for rewritten expressions, and assigning special traces to recursive strategies which can only be performed on the new base types. The trace computations also need to be updated to accommodate the special traces. This approach isolates the usage of recursion (to minimize its influence on current precisely traced type system) and avoids complex self-referential traces, but an obvious trade-off is the loss of completely precise tracing.

Generic Traversal

For generic traversal, as mentioned in Section 5.3.2, the underlying structural type system is not deeply tied with the tracing system, so it is possible to extend Core $S_t$ with limited forms of generic traversal by changing the underlying type system.
For example, by adding first-class labels from Leijen [2004] which parameterizes the labels in variant constructions and row types, the ad-hoc non-recursive traversal combinators defined in Example 5.9 can be generalized to work for arbitrary labels, and we can also define a generically labelled version of the \textbf{both} combinator as follows.

\begin{verbatim}
let labBoth = st xs => st ys =>
  rule l (x , y) -> ((x , y), l ());
  (xs & & ys) & & id;
  rule ((xr , yr), l ()) -> l (xr , yr):
  ∀ l : Lab. ∀ r : Row. (r \ l) ⇒
  (vx₁[β₀] ↦ vx₁[β₁]) ⇒ (vx₁[β₀] ↦ vx₁[β₁]) ⇒
  (⟨l : (vx₁[β₀] v yr₁[β₁] | r) | [α]⟩ ↦ ⟨l : (vx₁[β₀] v yr₁[β₁] | r)⟩)
\end{verbatim}

In this definition, we assume that the support of first-class labels as described by Leijen [2004] has been added to Core $S_t$, so the labels in patterns can be written as variables such as the $l$ in this definition. The rule in line 2 decomposes the labelled pair and remembers the actual label, and line 3 applies the parameter strategies ($xs$ and $ys$) while keeping the label unchanged, and finally the rule in line 4 reconstructs the labelled pair using the remembered label.

The more interesting part is the type of this definition. The strategy combinator type resembles the type of the \textbf{both} combinator, except that the pair types in the returned strategy type (line 8) are now labelled by $l$ and wrapped by variant types. Since labels are not traced, this change does not affect the tracing structure and the precision of tracing. Another significant change is the universally quantified type variables and the type predicate in line 6. The $Lab$ and $Row$ kinds and the ($r \ l$) predicate are adapted from Leijen [2004], where $Lab$ is the kind of labels, $Row$ is the kind of rows, and ($r \ l$) means the instantiation of $r$ cannot contain the instantiation of $l$. The row kind in Core $S_t$ is not used here because the row kind $R$ works as a constraint describing the constant labels that should be absent in a row, but $R$ is not expressive enough to support variable labels. For first-class labels, as explained in Leijen [2004], constraining rows with predicates and performing type inference under the qualified type framework is a more suitable choice. Thus, to introduce first-class labels, the underlying type system of Core $S_t$ needs to be reformulated under the qualified type framework.

Compared with the generic traversal combinator \texttt{all}, the definition of \texttt{labBoth} is
still not generic enough. Although it traverses the elements of a labelled pair regardless of the label, \texttt{labBoth} can only perform this traversal for labelled pairs instead of data structures with arbitrarily many fields. This is actually a common restriction for all three type systems in this thesis, so more detailed discussion is in Chapter 6. Besides, \texttt{labBoth} requires two parameter strategies, \texttt{x} and \texttt{y}, to rewrite the two elements of the pair, but for the \texttt{all} combinator, only one parameter strategy is needed to perform the rewriting of all sub-terms. This difference is caused by the linear restriction for strategy variables in Core $S_t$, and a deeper reason is that typing non-linear usage of strategy variables violates the precise tracing assumption.

Finally, to support fully-fledged generic traversals with precise tracing, traceable recursion is needed, but as discussed above, there is a persistent conflict between recursive definitions and the tracing mechanism in Core $S_t$. Thus, generic traversal is not supported in Core $S_t$.

**Robustness of Formal Guarantees to Type System Extensions**

It is worth noting that the statements about well-typed but empty-traced strategies are more resilient to changes in the type system, as they only rely on type soundness and strong normalization. Changes such as switching to a deterministic version of choice, allowing untraced fallible conditions in rewrite rule definitions and adding base types which cannot be precisely traced (for example, the expression types for recursive strategies mentioned above), can make the statements about the success of well-typed and well-traced strategy executions no longer hold, but as long as the type system is sound and strong normalizing, we can still state that the execution result of well-typed but empty-traced strategies must be \texttt{fail}.

**5.5 Extensions**

This section introduces three possible extensions for Core $S_t$: integer pattern, rewrite condition, and conditional rewrite, and discuss their influence on the formal guarantees, respectively.

**5.5.1 Integer Pattern**

Besides representing natural numbers as labels as shown in Section 5.3.1, it is possible to extend the language with integer patterns as follows, where \cdots represents the
The types are extended with the integer type $\text{Int}$, which is a traceable type.

$$\tau := \cdots | \text{Int}[\psi]$$

Noticeably, the integer type can be traced by a trace variable set $\psi$ like the unit type, because besides the unit pattern and the unit type, the integer pattern and the integer type now becomes another non-variable terminal in the pattern and type syntax, respectively, allowing the integer type to be directly traced by trace identifiers.

Example 5.1 would be written as follows if the integer patterns are used instead of the natural number labels.

```verbatim
let e1Int = rule m * n -> n * m ;
  (rule #1 * v -> v || rule #2 * w -> w + w) :
  d0 * Int | e0 * Int $\xrightarrow{[d,e]}$ d0 | e0 + e0
```

Figure 5.19 collects all the kinding, tracing and typing rule extensions required by the integer pattern extension. All the rules are similar to those for the unit type. For other definitions such as $\text{erase}$ and $\text{unifyTrace}$, the treatment of the integer type case is the same as the unit type case.

For the operational semantics, the values are extended with the integer value.

$$\text{Values} := \cdots | \#\mathbb{Z}$$
Importantly, the match function should also be extended with the following case for the integer patterns.

\[ \text{match}(\#n; \#n) = (\cdot \mapsto \cdot) \]

With this change, successfully unified traces no longer guarantee successful pattern matchings because different integers do not match with each other while they have the same \textbf{Int} type. As a consequence, we are not able to prove Lemma 5.3.10 and all other properties depending on Lemma 5.3.10, that is, with the integer pattern extension, we cannot say well-typed and well-traced strategy executions must succeed.

On the other hand, as explained in Section 5.4.2, the statements about well-typed but empty-traced strategies still hold.

Following a similar routine, more base type extensions can be added, which can encapsulate values with complex internal structures and avoid overly complicated structural types. Meanwhile, the drawback is the loss of precise tracing and the related guarantees for well-typed and well-traced strategies.

### 5.5.2 Rewrite Condition

Although the rewrite condition and the corresponding untraced sub-language are removed in Core $S_t$ for stronger formal guarantees and simpler formalization, they can be brought back to support more complex rewrites. According to $\textbf{P-Var}$, the type of a pattern variable in a rewrite rule definition can have an arbitrary underlying type (which of course should be well-formed and traceable), and by the tracing mechanism for pattern variables, the attached traces can be easily erased. Thus, in a rewrite rule definition, it is safe and convenient to erase the traces in the type of pattern variables and put them into a rewrite condition environment, where the untraced sub-language can perform further analysis and produce a Boolean rewrite condition. Chapter 4 has already shown how to include an untraced sub-language and how to typecheck and evaluate rewrite rules with rewrite condition, and the support of rewrite condition can be similarly done in Core $S_t$, so the details are not discussed here.

It is worth noting that, as mentioned in Section 5.4.2 and Section 4.6, rewrite condition makes the statements about the success of well-typed and well-traced strategy executions no longer hold. Specifically, allowing rewrite conditions to decide the success or failure of rewrite rules breaks Lemma 5.3.10 and related theorems. This is a drawback of supporting rewrite condition. On the other hand, the statements about well-typed but empty-traced strategies still hold.
5.5.3 Conditional Rewrite

To avoid any potential confusion, rewrite condition introduced above is an extra Boolean condition in rewrite rule definition, which can decide the success or failure of the rule, but conditional rewrite is a composition of three strategies, forming an \texttt{if cs then ts else es} structure. The code below defines the conditional combinator, and this definition requires the choice combinator is left-biased.

\begin{verbatim}
let cond = st cs => st ts => st es =>
    (rule i -> (i, i) ; cs && id ;
    rule (r, i) -> i ; ts) || es :
    (a_0 \rightarrow a_1) \Rightarrow (a_0 \rightarrow a_2) \Rightarrow (b_0 \rightarrow b_1) \Rightarrow (a_0 | b_0 \rightarrow a_2 | b_1)
\end{verbatim}

The condition strategy \texttt{cs} is firstly run on the input, if \texttt{cs} succeeds, the then branch strategy \texttt{ts} is executed on the input, otherwise the else branch strategy \texttt{es} is executed on the input. This definition puts \texttt{cs} and \texttt{ts} into a sequential composition, so the execution of \texttt{ts} depends on the execution result of \texttt{cs}. It also makes use of the \texttt{&&} combinator to maintain a copy of the original input \texttt{i}, so all three parameter strategies have access to \texttt{i}. Besides, \texttt{cs} must be executed before \texttt{es}, so the left-biased choice combinator is necessary to ensure the strict order of execution.

The traced type of \texttt{cond} shows two paths of execution, identified by \texttt{a} and \texttt{b}, respectively. The trace identified by \texttt{a} describes the dependent executions of \texttt{cs} and \texttt{ts}. An interesting observation is that, without supplying any actual input, if none of the expected input shapes of \texttt{cs} matches those of \texttt{ts}, trace \texttt{a} will be removed, indicating that for any future compositions, they should not expect to get any successful results from \texttt{ts}, because whatever the input is, either \texttt{cs} or \texttt{ts} would fail due to mismatched input shapes. On the other hand, the trace identified by \texttt{b} describe the execution of \texttt{es}. However, with the use of left-biased choice, the type of \texttt{cond} does not have completely precise traces and does not show that the execution of \texttt{es} also depends on the execution result of \texttt{cs}.

5.6 Conclusion

In this chapter, we present Core \texttt{St}, a strategy language with a trace typed system detecting ill-composed strategies. Compared with Rewrite \texttt{St}, it simplifies the formalization, provides stronger formal guarantees, and allows more flexible strategy compositions.
We discuss a number of exemplar strategy compositions, demonstrating the capabilities of our type system to compute precise types for strategy compositions as well as raising warnings and errors for problematic strategy compositions.

We confirm the soundness of our type system, formally justify why we reject empty-traced strategies (as they are guaranteed to fail at runtime), and show that well-traced strategies must have a possible execution path at runtime. Finally, we discuss the design choices we have made in Core $S_t$ and the corresponding trade-offs, and possible extensions.
Appendix 5.A  Definition of trace operations

\[
\text{Select } \varphi \text{ in } (\Phi; \Sigma; \omega) = (\Phi; \Sigma; \omega)
\]

for \( \alpha \) in \( \varphi \)

\[
\begin{align*}
\text{if } \alpha \triangleright \phi & \in \Phi \\
\Sigma_i & := \\
\text{for } (x : \omega_x) \text{ in } \Sigma \\
\Sigma_i & := \Sigma_i, x : \text{selectInType}(\Phi; \alpha; \omega_x)
\end{align*}
\]

\[
(\Phi_r; \Sigma_r; \omega_r) := \text{Add } (\alpha \triangleright \phi; \Sigma_i; \text{selectInType}(\Phi; \alpha; \omega)) \text{ to } (\Phi_r; \Sigma_r; \omega_r)
\]

return \( (\Phi_r; \Sigma_r; \omega_r) \)

Figure 5.20: Definition of Select \(-\) in \(-\) where selectInType is defined in Figure 5.21
selectInType : (Φ; α; t) → t
selectInType(α ▷ φ, Φ_r; α; ([])_{α, ψ}) = ()_{α}
selectInType(α ▷ [β, φ_r], Φ_r; α; τ_{β, ψ_r}) = erase(τ_{β})
selectInType(α ▷ φ, Φ_r; α; τ_p) = selectInType(α ▷ φ, Φ_r; α; τ) if φ ∩ ψ = Ø and α ∉ ψ
selectInType(Φ; α; ⟨ρ⟩) = ⟨selectInType(Φ; α; ρ)⟩
selectInType(Φ; α; v as ⟨ρ⟩) = v as (selectInType(Φ; α; ρ))
selectInType(Φ; α; ·) = ·
selectInType(Φ; α; ε) = ε
selectInType(Φ; α; (ℓ : τ | ρ)) = (ℓ : selectInType(Φ; α; τ) | selectInType(Φ; α; ρ))
selectInType(Φ; α; ()) = ()
selectInType(Φ; α; (τ_m, τ_n)) = (selectInType(Φ; α; τ_m), selectInType(Φ; α; τ_n))
selectInType(Φ; α; [α, ϕ_r] ▷ τ) = [α] ▷ selectInType(Φ; α; τ)
selectInType(Φ; α; ϕ ▷ τ) = [] ▷ erase(τ) if α ∉ ϕ
selectInType(Φ; α; τ_p [α, ϕ_r] → τ_e) =
selectInType(Φ; α; τ_p → [α] selectInType(Φ; α; τ_e)
selectInType(Φ; α; τ_p → [α] erase(τ_p) [] erase(τ_e) if α ∉ ϕ
selectInType(Φ; α; (τ_p → τ_e) ⇒ ω) =
selectInType(Φ; α; τ_p → τ_e) ⇒ selectInType(Φ; α; ω)
Figure 5.22: Definition of Fresh –
Add $\to$ to $(\Phi; \Sigma; \omega) \to (\Phi; \Sigma; \omega) \to (\Phi; \Sigma; \omega)$

Add $(\Phi_m; \Sigma_m; \omega_m)$ to $(\Phi_n; \Sigma_n; \omega_n) = $

$\Sigma_r := \cdot$

for $(x : \omega_x)$ in $\Sigma_m$

$\Sigma_r := \Sigma_r, x : addType(\Sigma_m(x); \Sigma_n(x))$

return $(\Phi_m, \Phi_n, \Sigma_r; addType(\omega_m; \omega_n))$

addType : $(t; t) \rightarrow t$

addType$(\tau^m_\psi; \tau^n_\psi) = addType(\tau^m_\psi; \tau^n_\psi)_{[\psi_m; \psi_n]}$

addType$(\tau^m_\psi; \tau^n_\psi) = addType(\tau^m_\psi; \tau^n_\psi)_\psi$

addType$(\tau^m_\psi; \tau^n_\psi) = addType(\tau^m_\psi; \tau^n_\psi)_\psi$

addType$(\langle \rho_m; \rangle; \langle \rho_n; \rangle) = \langle addType(\rho_m; \rho_n) \rangle$

addType$(\nu as \langle \rho_m; \rangle; \nu as \langle \rho_n; \rangle) = \nu as \langle addType(\rho_m; \rho_n) \rangle$

addType$(\cdot; \cdot) = \cdot$

addType$(\nu; \nu) = \nu$

addType$(\ell : \tau_m | \rho_m); (\ell : \tau_n | \rho_n)) = (\ell : addType(\tau_m; \tau_n) | addType(\rho_m; \rho_n))$

addType$((); ()) = ()$

addType$(\tau_m, \tau_n); (\tau_j, \tau_k)) = (addType(\tau_m; \tau_j), addType(\tau_n; \tau_j))$

addType$(\varphi_m \triangleright \tau_m; \varphi_n \triangleright \tau_n) = [\varphi_m, \varphi_n] \triangleright addType(\tau_m; \tau_n)$

addType$(\tau^m_\phi \rightarrow \tau_n; \tau^j_\phi \rightarrow \tau_k) = addType(\tau^m_\phi; \tau^j_\phi) \triangleright addType(\tau_n; \tau_k)$

addType$(\tau^m_\phi \rightarrow \tau_n) \Rightarrow \omega_m; (\tau^j_\phi \rightarrow \tau_k) \Rightarrow \omega_j) =$

addType$(\tau^m_\phi \rightarrow \tau_n; \tau^j_\phi \rightarrow \tau_k) \Rightarrow addType(\omega_m; \omega_j)$

Figure 5.23: Definition of Add $\to$ −
Appendix 5.B  Proofs

In the following proofs, we write \(\text{mems}(\Phi; \omega)\) (defined in Figure 5.24) for the set of types traced by a trace member, extracted from a single-traced type \(\omega\) in the tracing environment \(\Phi\). The parameter \(\Phi\) is omitted if it is clear from the context. It can be easily generalized to be used on environments.

\[
\begin{align*}
\text{mems} &: (\Phi; t) \rightarrow \{\tau\} \\
\text{mems}(\alpha \triangleright [\beta, \phi]; \tau_{[\beta]}) &= \{\tau_{[\beta]}\} \\
\text{mems}(\Phi; \nu) &= \emptyset \\
\text{mems}(\Phi; \langle \rho \rangle) &= \text{mems}(\Phi; \rho) \\
\text{mems}(\Phi; \cdot) &= \emptyset \\
\text{mems}(\Phi; (\tau_m, \tau_n)) &= \text{mems}(\Phi; \tau_m) \cup \text{mems}(\Phi; \tau_n) \\
\text{mems}(\Phi; \cdot) &= \emptyset \\
\text{mems}(\Phi; (\ell : \tau | \rho)) &= \text{mems}(\Phi; \tau) \cup \text{mems}(\Phi; \rho) \\
\text{mems}(\Phi; \phi \triangleright \tau) &= \text{mems}(\Phi; \tau) \\
\text{mems}(\Phi; \phi \triangleright \tau) &= \text{mems}(\Phi; \tau) \cup \text{mems}(\Phi; \omega) \\
\end{align*}
\]

Figure 5.24: Definition of mems

**Lemma 5.3.4** (Rule Substitution). If \(\Delta; \Gamma \vdash (\text{rule } p \rightarrow e) \leftarrow \text{succ } v : [\alpha_r] \triangleright [\tau_r \triangleleft \alpha_r | []]\), and \(\text{match}(p; v) = S \neq \text{failPat}\), then \(\Delta; \Gamma \vdash \text{succ } e[S] : [\alpha_r] \triangleright [\tau_r \triangleleft \alpha_r | []]\).

*Proof of Lemma 5.3.4.* By inversion on \(\text{T-Exec}\) and \(\text{T-R-Lam}\), we know \(\Delta \vdash_{\alpha_f} p : \tau_p \triangleright \Phi; \Theta, \text{ and } \Delta; \Gamma; \Phi; \Theta \vdash e : \tau_e\), and \(\Delta; \Gamma; \alpha_e \triangleright [\cdot] ; \star \vdash v : \tau_x\), and \(\text{erase}(\tau_p) = \text{erase}(\tau_e)\), and there is a \(\overline{S}\) generated by \(\text{unifyTrace}(\alpha_f \triangleright \phi_f; [\alpha_f] \triangleright \tau_p; [\alpha_f] \triangleright (\tau_x[\alpha_o \mapsto \alpha_f]))\). By induction on the typing derivation of \(\Delta \vdash_{\alpha_f} p : \tau_p \triangleright \Phi; \Theta, \text{ and with } \text{match}(p; v) \neq \text{failPat}\) and the definition of \(\text{unifyTrace}\), we know that \(\overline{S} \neq \text{failTrace}\). Then by Lemma 5.3.2 and Lemma 5.3.3, we know that \(\tau_o[\alpha_o \mapsto \alpha_f] = \tau_p[\overline{S}]\), and for each variable \(x\) in \(p\) and also in \(\text{dom}(S)\), there is a corresponding \(\tau_{[\beta]}\) in \(\tau_p\) and also in \(\text{dom}(\overline{S})\), and \(\Delta; \Gamma; \alpha_f \triangleright [\cdot] ; \star \vdash S(x) : \overline{S}(\tau_{[\beta]})\). Then it is easy to show that \(\Delta; \Gamma; \alpha_f \triangleright [\cdot] ; \star \vdash e[S] : \tau_e[\overline{S}]\), which with appropriate renaming provides evidence for the proof goal. \(\Box\)
Lemma 5.3.5 (Failed Rule). If $\Delta;\Gamma \vdash (\text{rule } p \rightarrow e) \leftarrow \text{succ } v : [] \triangleright \tau_r$, then $(\text{rule } p \rightarrow e) \leftarrow \text{succ } v \leadsto \text{fail}.$

Proof of Lemma 5.3.5. By inversion on T-Exec and T-R-Lam, we know $\text{failTrace} = \text{unifyTrace}(\alpha_f \triangleright \phi_f; [\alpha_f] \triangleright \tau_p; [\alpha_f] \triangleright (\tau_0[\alpha_0 \mapsto \alpha_f])), \text{ where } \Delta \vdash_{\alpha_f} p : \tau_p \vdash \Phi; \Theta, \text{ and } \Delta;\Gamma;\Phi;\Theta \vdash e : \tau_e, \text{ and } \Delta;\Gamma;\alpha_0 \triangleright []; \star \vdash v : \tau_0, \text{ and } \text{erase}(\tau_p) = \text{erase}(\tau_e).$ By induction on the typing derivation of $\Delta \vdash_{\alpha_f} p : \tau_p \vdash \Phi; \Theta$ and the definition of $\text{unifyTrace}$, we know that the only source of $\text{failTrace}$ is mismatched labels in $p$ and $v$. By the definition of match, we know the result of $\text{match}(p; v)$ must be $\text{failPat}$. Thus, $(\text{rule } p \rightarrow e) \leftarrow \text{succ } v \leadsto \text{fail}. \quad \Box$

Lemma 5.3.6 (Strategy Substitution). If $\Delta;\Gamma;\Phi^\downarrow;\Sigma^\downarrow \vdash (\text{st } x \Rightarrow e_f) \leftarrow e : \omega + \Phi^\downarrow;\Sigma^\uparrow,$ then $\Delta;\Gamma;\Phi^\downarrow;\Sigma^\downarrow \vdash e_f[x \mapsto e] : \omega + \Phi^\downarrow;\Sigma^\uparrow.$

Proof of Lemma 5.3.6. By inversion on T-S-App and T-S-Lam, we know that:  

\[
\begin{align*}
\Delta;\Gamma;\Phi^\downarrow;\Phi^\downarrow_1;\Sigma^\downarrow_1; x : \omega^\downarrow_x + e_f : \Phi_f + \Phi^\downarrow_1;\Sigma^\downarrow_1, x : \omega^\downarrow_x & \quad \Delta;\Gamma;\Phi^\downarrow;\Sigma^\downarrow + e : \omega_e + \Phi^\downarrow_1;\Sigma^\downarrow_e \\
\text{erase}(\omega^\downarrow_x) = \text{erase}(\omega_e) & \quad (\Phi^\downarrow_1;\Sigma^\uparrow_1;\omega^\downarrow_e) = \text{compTrace}(\Phi^\downarrow_1;\Sigma^\uparrow_1;\Phi^\downarrow_1;\Sigma^\uparrow_1;\omega^\downarrow_e)
\end{align*}
\]

Our goal is to show that $\Delta;\Gamma;\Phi^\downarrow;\Sigma^\downarrow + e_f[x \mapsto e] : \omega + \Phi^\downarrow;\Sigma^\uparrow.$ The proof proceeds by induction on the typing derivation of $\Delta;\Gamma;\Phi^\downarrow;\Phi^\downarrow_1;\Sigma^\downarrow_1; x : \omega^\downarrow_x + e_f : \Phi_f + \Phi^\downarrow_1;\Sigma^\downarrow_1, x : \omega^\downarrow_x$:

- **Case** $e_f = x$

  This case is trivial by T-S-Var.

- **Case** $e_f = e_g \leftarrow e_g$

  ★ **When** $x$ is linearly used in $e_g$, by inversion on T-S-App, we know that:

  \[
  \begin{align*}
  \Delta;\Gamma;\Phi^\downarrow;\Phi^\downarrow_1;\Sigma^\downarrow_1, x : \omega^\downarrow_x + e_g : \omega_g & \Rightarrow \omega_h + \Phi^\downarrow_1;\Sigma^\downarrow_g \\
  \Delta;\Gamma;\Phi^\downarrow;\Phi^\downarrow_1;\Sigma^\downarrow_1, x : \omega^\downarrow_x + e_g : \omega_g + \Phi^\downarrow_1;\Sigma^\downarrow_1; x : \omega^\downarrow_x \\
  \text{erase}(\omega_g) & \quad (\Phi^\downarrow_1;\Sigma^\downarrow_1; x : \omega^\downarrow_x;\omega_f) = \text{compTrace}(\Phi^\downarrow_1;\Sigma^\downarrow_1;\omega^\downarrow_g;\omega_g;\Phi^\downarrow_1;\Sigma^\downarrow_1, x : \omega^\downarrow_g)
  \end{align*}
  \]

  By the induction hypothesis we know that:

  \[
  (\Phi^\downarrow_1;\Sigma^\downarrow_1;\omega^\downarrow_g) = \text{compTrace}(\Phi^\downarrow_1;\Sigma^\downarrow_1;\omega^\downarrow_g;\omega_g;\Phi^\downarrow_1;\Sigma^\downarrow_1;\omega^\downarrow_g) \\
  \Delta;\Gamma;\Phi^\downarrow;\Phi^\downarrow_1;\Sigma^\downarrow_1 + e_g[x \mapsto e] : \omega^\downarrow_g + \Phi^\downarrow_1;\Sigma^\downarrow_1
  \]

  Our goal is to show that $(\Phi^\downarrow_1;\Sigma^\uparrow_1;\omega^\downarrow_e) = \text{compTrace}(\Phi^\downarrow_1;\Sigma^\downarrow_1;\omega^\downarrow_g;\omega_g;\Phi^\downarrow_1;\Sigma^\downarrow_1;\omega^\downarrow_g)$. Since the operation in $\text{compTrace}$ is uniform for each trace during the enumeration, it is sufficient to consider a single trace (up to renaming) for the proof. We use a tilde on the name ($\tilde{\omega}_e$, for example) to indicate that only a single trace is considered.
* For each trace which is present in $\Phi^t$, listed below is the evidence we currently know:

\[
\begin{align*}
\hat{\omega}_e[\bar{S}_e] &:= \hat{\omega}_e^y[\bar{S}_e] & \hat{\omega}_f[\bar{S}_e] &= \hat{\omega} & (\hat{\Sigma}_f, \hat{\Sigma}_e)[\bar{S}_e] &= \hat{\Sigma}_e \uparrow \\
\hat{\omega}_g[\bar{S}_g] &:= \hat{\omega}_g[\bar{S}_g] & \hat{\omega}_h[\bar{S}_g] &= \hat{\omega}_f & \hat{\omega}_x[\bar{S}_g] &= \hat{\omega}_x & (\hat{\Sigma}_g, \hat{\Sigma}_x)[\bar{S}_g] &= \hat{\Sigma}_g \uparrow \\
\hat{\omega}_e[\bar{S}_g] &:= \hat{\omega}_e^y[\bar{S}_g] & \hat{\omega}_y[\bar{S}_g] &= \hat{\omega}_y & (\hat{\Sigma}_y, \hat{\Sigma}_e)[\bar{S}_g] &= \hat{\Sigma}_y \uparrow \\
\end{align*}
\]

We write $\hat{=}:$ in some equations to indicate that the equation also defines a substitution. For example, $\hat{\omega}_e[\bar{S}_e] := \hat{\omega}_e^y[\bar{S}_e]$ means $\bar{S}_e$ is defined by the trace unification of $\hat{\omega}_e$ and $\hat{\omega}_e^y$ using $\text{unifyTrace}$. We need to show that $\text{unifyTrace}$ gives us a substitution $\bar{S}_h$ (if the unification is successful), such that $\hat{\omega}_g[\bar{S}_h] := \hat{\omega}_g^y[\bar{S}_h$, $\hat{\omega}_h[\bar{S}_h] = \hat{\omega}_h$, and $(\hat{\Sigma}_h, \hat{\Sigma}_e)[\bar{S}_h] = \hat{\Sigma}_e \uparrow$, that is:

\[
\begin{align*}
\hat{\omega}_g[\bar{S}_h] &:= \hat{\omega}_g[\bar{S}_g] \mid [\bar{S}_e] & \hat{\omega}_h[\bar{S}_h] &= \hat{\omega}_h[\bar{S}_g] \mid [\bar{S}_e] \\
(\hat{\Sigma}_g, (\hat{\Sigma}_y, \hat{\Sigma}_e))[\bar{S}_g] &\mid [\bar{S}_e] = ((\hat{\Sigma}_g^y, \hat{\Sigma}_y), \hat{\Sigma}_e)[\bar{S}_e] \\
\end{align*}
\]

- For showing that $\hat{\omega}_h[\bar{S}_h] = \hat{\omega}_h[\bar{S}_g] \mid [\bar{S}_e]$ with $\bar{S}_h$ defined as

\[
\hat{\omega}_g[\bar{S}_h] := \hat{\omega}_g[\bar{S}_g] \mid [\bar{S}_e],
\]

we need to look into the definition of $\bar{S}_h$ and $\bar{S}_e$.

- For the definition of $\bar{S}_h$, we know that:

\[
\begin{align*}
dom(\bar{S}_g) &\subseteq \text{mems}(\hat{\omega}_e) \cup \text{mems}(\hat{\omega}_e^y) \\
\text{mems}(\hat{\omega}_e) \cap \text{mems}(\hat{\omega}_e^y) &= \emptyset \\
dom(\bar{S}_g) &\subseteq \text{mems}(\hat{\omega}_g) \cup \text{mems}(\hat{\omega}_y) \\
\text{mems}(\hat{\omega}_g) \cap \text{mems}(\hat{\omega}_g^y) &= \emptyset \\
\text{mems}(\hat{\omega}_g) \cap \text{mems}(\hat{\omega}_e) &= \emptyset \quad \text{mems}(\hat{\omega}_g) \cap \text{mems}(\hat{\omega}_e^y) = \emptyset \\
\text{mems}(\hat{\omega}_g) \cap \text{mems}(\hat{\omega}_e^y) &\supseteq \emptyset \\
\end{align*}
\]

Thus, $\text{mems}(\bar{S}_g(\text{dom}(\bar{S}_g) \cap \text{dom}(\bar{S}_g))) \cap (\text{dom}(\bar{S}_g) \cup \text{mems}(\hat{\omega}_g)) = \emptyset$, and by the definition of $\text{unifyTrace}$ we can split $\bar{S}_h$ to two stages, named as $\bar{S}_h^{gg}$ and $\bar{S}_h^u$, where $\bar{S}_h^{gg}$ can be defined as follows based on $\bar{S}_g$.

$\bar{S}_h^{gg} = \forall \tau[\beta] \in \text{dom}(\bar{S}_g) \setminus \text{dom}(\bar{S}_g), \tau[\beta] \mapsto \bar{S}_g(\tau[\beta])$.

As for $\bar{S}_h^u$, we can deduce that $\text{dom}(\bar{S}_h^u) \subseteq \text{mems}(\bar{S}_g(\text{dom}(\bar{S}_g) \cap \text{dom}(\bar{S}_g))) \cup \text{mems}(\bar{S}_g(\text{dom}(\bar{S}_g) \cap \text{dom}(\bar{S}_g)))$, and for all $\tau[\beta] \in \text{dom}(\bar{S}_h^u)$, $\bar{S}_g(\tau[\beta]) \mid [\bar{S}_h^u] = \bar{S}_g(\tau[\beta]) \mid [\bar{S}_h^u]$.

Finally, we get $\hat{\omega}_g[\bar{S}_h] \mid [\bar{S}_h] = \hat{\omega}_g[\bar{S}_g] \mid [\bar{S}_h]$. 

- For the definition of $\bar{S}_e$, from the evidence we know that
\( \dot{\omega}_e[\bar{S}_e] := \dot{\omega}_x[\bar{S}_g][\bar{S}_e] \). Similarly, we have
\[
\text{mems}(\bar{S}_g(\text{dom}(\bar{S}_g) \cap \text{dom}(\bar{S}_y))) \cap (\text{dom}(\bar{S}_y) \cup \text{mems}(\dot{\omega}_y)) = \emptyset,
\]
and by the definition of \textit{unifyTrace} we can split \( \bar{S}_e \) to two stages, named as \( \bar{S}_e^g \) and \( \bar{S}_e^u \), where \( \bar{S}_e^g \) can be defined as follows based on \( \bar{S}_y \).
\[
\bar{S}_e^g = \forall \tau_{[\beta]} \in \text{dom}(\bar{S}_y) \setminus \text{dom}(\bar{S}_g), \tau_{[\beta]} \mapsto \bar{S}_g(\tau_{[\beta]}).
\]
As for \( \bar{S}_e^u \), we can deduce that \( \text{dom}(\bar{S}_e^u) \subseteq \text{mems}(\bar{S}_y(\text{dom}(\bar{S}_g) \cap \text{dom}(\bar{S}_y))) \cap \text{mems}(\bar{S}_g(\text{dom}(\bar{S}_g) \cap \text{dom}(\bar{S}_y))) \)
and for all \( \tau_{[\beta]} \in \text{dom}(\bar{S}_e^u), \bar{S}_g(\tau_{[\beta]})[\bar{S}_e^u] = \bar{S}_g(\tau_{[\beta]})[\bar{S}_e^u] \).
Finally, we get \( \dot{\omega}_e[\bar{S}_e^u] = \dot{\omega}_x[\bar{S}_g][\bar{S}_e^u] \).

Now we need to show that \( \dot{\omega}_h[\bar{S}_h^g][\bar{S}_h^u] = \dot{\omega}_h[\bar{S}_g][\bar{S}_e^g][\bar{S}_e^u] \).

Since \( \text{mems}(\dot{\omega}_h) \cap \text{dom}(\bar{S}_y) = \emptyset, \bar{S}_h^g \) can be broken down, and we get \( \dot{\omega}_h[\bar{S}_g][\bar{S}_y][\bar{S}_h^u] \). On the other hand, since \( \text{mems}(\dot{\omega}_h[\bar{S}_g]) \cap \text{dom}(\bar{S}_y) = \emptyset \) and \( \text{mems}(\dot{\omega}_h[\bar{S}_g][\bar{S}_y]) \cap \text{dom}(\bar{S}_y) = \emptyset \), \( \bar{S}_g^g \) can be replaced by \( \bar{S}_y \), and we get \( \dot{\omega}_h[\bar{S}_g][\bar{S}_y][\bar{S}_y^u] \). Finally, \( \bar{S}_h^g \) and \( \bar{S}_e^u \) are identical by their definitions, so we get the equation
\[
\dot{\omega}_h[\bar{S}_g][\bar{S}_y][\bar{S}_h^u] = \dot{\omega}_h[\bar{S}_g][\bar{S}_y][\bar{S}_e^u],
\]
that is, \( \dot{\omega}_h[\bar{S}_h^u] = \dot{\omega}_h[\bar{S}_g][\bar{S}_e^u] \).

For showing that \((\hat{\Sigma}_y^g, \hat{\Sigma}_y^u, \hat{\Sigma}_y^u)[\bar{S}_y]) \cap \bar{S}_h^u = ((\hat{\Sigma}_y^g, \hat{\Sigma}_y^u, \hat{\Sigma}_y^u)[\bar{S}_g], \bar{\Sigma}_e)[\bar{S}_e] \),
we can decompose the equation into three parts as follows.

- \( \hat{\Sigma}_y^g[\bar{S}_h^u] = \hat{\Sigma}_y^g[\bar{S}_g][\bar{S}_e] \) can be analysed in a similar way to \( \dot{\omega}_h[\bar{S}_h^u] = \dot{\omega}_h[\bar{S}_g][\bar{S}_e] \).

- \( \hat{\Sigma}_y^u[\bar{S}_y][\bar{S}_h^u] = \hat{\Sigma}_y^u[\bar{S}_g][\bar{S}_h^u] \) can be written as \( \bar{S}_y^g \circ \bar{S}_y^g \), where \( \bar{S}_y^g = \forall \tau_{[\beta]} \in \text{dom}(\bar{S}_y) \cap \text{dom}(\bar{S}_g), \tau_{[\beta]} \mapsto \bar{S}_y(\tau_{[\beta]}) \) and \( \bar{S}_y^g = \forall \tau_{[\beta]} \in \text{dom}(\bar{S}_y) \setminus \text{dom}(\bar{S}_g), \tau_{[\beta]} \mapsto \bar{S}_g(\tau_{[\beta]}) \) and similarly representing \( \bar{S}_y \) as \( \bar{S}_g^g \circ \bar{S}_y^g \), we get \( \hat{\Sigma}_y^u[\bar{S}_y][\bar{S}_y^g][\bar{S}_y^u] = \hat{\Sigma}_y^u[\bar{S}_y][\bar{S}_g^g][\bar{S}_y^u] \).

For \( \bar{S}_h^g \cap \text{mems}(\hat{\Sigma}_y^u) = \text{dom}(\bar{S}_h^g) \cap \text{mems}(\hat{\Sigma}_y^u)[\bar{S}_g] \).

Besides, for all \( \tau_{[\beta]} \in (\text{dom}(\bar{S}_g) \setminus \text{dom}(\bar{S}_y)) \cap \text{mems}(\hat{\Sigma}_y^u)[\bar{S}_g] \), there is \( \bar{S}_y(\bar{S}_g(\tau_{[\beta]})) = \bar{S}_g(\tau_{[\beta]}) \) and \( \text{mems}(\bar{S}_g(\tau_{[\beta]})) \cap \text{dom}(\bar{S}_y) = \emptyset \).

Thus, \( \bar{S}_h^g \) here can be replaced by \( \bar{S}_g \) and its position can be swapped with \( \bar{S}_y^g \) and \( \bar{S}_y^u \). We can do similarly for \( \bar{S}_e^g \) and replace it with \( \bar{S}_y^g \).
Then we get $\mathcal{S}_y[\mathcal{S}_y][\mathcal{S}_y][\mathcal{S}_h] = \mathcal{S}_y[\mathcal{S}_y][\mathcal{S}_y][\mathcal{S}_e]$. By the definition of $\mathcal{S}_h$ and $\mathcal{S}_e$, this equation holds.

- $\mathcal{S}_y[\mathcal{S}_y][\mathcal{S}_h] = \mathcal{S}_e[\mathcal{S}_e]$ can be written as $\mathcal{S}_y[\mathcal{S}_y][\mathcal{S}_h][\mathcal{S}_h] = \mathcal{S}_y[\mathcal{S}_y][\mathcal{S}_e][\mathcal{S}_e]$, and it can be analysed in a similar way to $\omega_h[\mathcal{S}_h] = \omega_h[\mathcal{S}_y][\mathcal{S}_e]$.

* For each trace which is not present in $\Phi^l$ because in the corresponding compTrace iteration the unifyTrace function returns failTrace
  - If $\mathcal{S}_g$ succeeds but $\mathcal{S}_e$ fails, $\mathcal{S}_y$ can
    - Fail. In this case, the corresponding trace won’t exists.
    - Succeed. In this case, $\mathcal{S}_h$ must fail. If both $\mathcal{S}_g$ and $\mathcal{S}_y$ succeed, then it is possible to define $\mathcal{S}_y$ as part of $\mathcal{S}_e$, and the failure of $\mathcal{S}_e$ becomes the only reason for $\mathcal{S}_e$ to fail. If $\mathcal{S}_e$ fails, $\mathcal{S}_h$ which share the same definition with $\mathcal{S}_e$ will also fail. Thus, $\mathcal{S}_h$ which depends on $\mathcal{S}_e$ must fail.
  - If $\mathcal{S}_g$ fails, then $\mathcal{S}_h$ must fail because if $\mathcal{S}_g$ which unifies $\omega_g$ and $\omega_y$ fails, it is impossible for $\mathcal{S}_h$ which unifies $\omega_g$ and $\omega_y[\mathcal{S}_h]$ to succeed.

* When $x$ is linearly used in $e_g$, the proof is similar.

- Case $e_f = st \ y \Rightarrow e_b$

  We need to show that $\Delta;\Gamma;\Phi^l;\Sigma^l; (st \ y \Rightarrow e_b)[x \mapsto e] : \omega + \Phi^l;\Sigma^l$. By linearity, we know that $y \neq x$, $x$ is linearly used in $e_b$, and so is $y$. Thus, by inversion on T-S-Lam, we know that:

$$
\Delta;\Gamma;\Phi^l;\Sigma^l; x : \omega_x, y : \omega_y + e_b : \omega_b + \Phi^l;\Sigma^l; x : \omega_x, y : \omega_y
$$

$$(\Phi^l;\Sigma^l); \omega) = \text{compTrace}(\Phi^l,f^l;\Sigma^l; \omega_x, y : \omega_y)$$

and by the induction hypothesis we know that:

$$(\Phi^l;x^l; y : \omega_x;\omega_y) = \text{compTrace}((\Phi^l,f^l;\Sigma^l; y : \omega_y;\omega_x);\omega_x;\omega_y;\Phi^l;\Sigma^l;\omega_e)$$

$$\Delta;\Gamma;\Phi^l;\Sigma^l; y : \omega_y + e_b[x \mapsto e] : \omega_b + \Phi^l;\Sigma^l; y : \omega_y$$

By the definition of compTrace, we know that $\Phi^l = \Phi^l$, $\Sigma^l = \Sigma^l$, and $\omega_y \Rightarrow \omega_y^l = \omega$.

Then by T-S-Lam, we know that $\Delta;\Gamma;\Phi^l;\Sigma^l; (st \ y \Rightarrow e_b)[x \mapsto e] : \omega_y^l$ gives us the proof goal.

$\square$
Lemma 5.B.1 (Seq Reduction). If $\Delta; \Gamma \vdash (\text{seq} \iff e_m \iff e_j) \iff e_i : \varphi_r \triangleright \tau_r + \Phi_r$, then $\Delta; \Gamma \vdash e_j \iff (e_m \iff e_i) : \varphi_r \triangleright \tau_r + \Phi_r$.

Proof of Lemma 5.B.1. By inversion on $\text{T-Seq}$, $\text{T-S-App}$ and $\text{T-Exec}$, we know:

- $\Delta; \Gamma \vdash e_m : \tau_m \triangleright e_n + \Phi_m$
- $\Delta; \Gamma \vdash e_j : \tau_j \triangleright e_k + \Phi_j$
- $\Delta; \Gamma \vdash e_i : \varphi_i \triangleright \tau_i + \Phi_i \triangleright []$

The full expansion of the $\text{compTrace}$ computation is too complex to show. Alternatively, since the operation in $\text{compTrace}$ is uniform for each trace during the enumeration, it is sufficient to consider a single trace (up to renaming) for the proof. We use a tilde on the name ($\tilde{\tau}_m$, for example) to indicate that only a single trace is considered.

- **For** each trace which is present in $\Phi_r$, we have
  
  $$\tilde{\tau}_n[\overline{\mathcal{S}}_{jn}] := \tilde{\tau}_j[\overline{\mathcal{S}}_{jn}], \quad \tilde{\tau}_m[\overline{\mathcal{S}}_{jm}][\overline{\mathcal{S}}_{imjn}] := \tilde{\tau}_i[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] = \tilde{\tau}_r$$

  We write := in some equations to indicate that the equation also defines a substitution. For example, $\tilde{\tau}_n[\overline{\mathcal{S}}_{jn}] := \tilde{\tau}_j[\overline{\mathcal{S}}_{jn}]$ means $\overline{\mathcal{S}}_{jn}$ is defined by the trace unification of $\tilde{\tau}_n$ and $\tilde{\tau}_j$ using $\text{unifyTrace}$. According the evidence we know, there must be a $\overline{\mathcal{S}}_{im}$ such that $\tilde{\tau}_m[\overline{\mathcal{S}}_{im}] := \tilde{\tau}_i$ because if we have $\tilde{\tau}_m[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] := \tilde{\tau}_i$, by the definition of $\text{unifyTrace}$, we must be able to get such $\overline{\mathcal{S}}_{im}$. Furthermore, there must also be a $\overline{\mathcal{S}}_{nimj}$ such that $\tilde{\tau}_j[\overline{\mathcal{S}}_{nimj}] := \tilde{\tau}_n[\overline{\mathcal{S}}_{im}]$ because we already know that $\tilde{\tau}_m[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] = \tilde{\tau}_i = \tilde{\tau}_m[\overline{\mathcal{S}}_{im}]$, and we can get $\tilde{\tau}_n[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] = \tilde{\tau}_n[\overline{\mathcal{S}}_{im}]$ because $\text{mems}(\tilde{\tau}_n) \subseteq \text{mems}(\tilde{\tau}_m)$, then by applying $\overline{\mathcal{S}}_{imjn}$ on both sides of $\tilde{\tau}_n[\overline{\mathcal{S}}_{jn}] = \tilde{\tau}_j[\overline{\mathcal{S}}_{jn}]$, we get $\tilde{\tau}_n[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] = \tilde{\tau}_j[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}]$, and we get $\tilde{\tau}_j[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] = \tilde{\tau}_j[\overline{\mathcal{S}}_{nimj}]$ by transitivity, and finally we get $\tilde{\tau}_k[\overline{\mathcal{S}}_{jn}][\overline{\mathcal{S}}_{imjn}] = \tilde{\tau}_k[\overline{\mathcal{S}}_{nimj}]$ because $\text{mems}(\tilde{\tau}_k) \subseteq \text{mems}(\tilde{\tau}_j)$. This also gives us the proof goal, that is $\tilde{\tau}_k[\overline{\mathcal{S}}_{nimj}] = \tilde{\tau}_r$.

- **For** each trace which is not present in $\Phi^\uparrow$ because in the corresponding $\text{compTrace}$ iteration the $\text{unifyTrace}$ function returns $\text{failTrace}$.

  - **If** $\overline{\mathcal{S}}_{jn}$ succeeds and $\overline{\mathcal{S}}_{imjn}$ fails, then $\overline{\mathcal{S}}_{im}$ can
    - **Fail.** In this case, the corresponding trace won’t exists.
    - **Succeed.** This case is only possible when $\overline{\mathcal{S}}_{imjn}$ fails because for each $\tau[\beta] \in \text{mems}(\tilde{\tau}_m)$, $\overline{\mathcal{S}}_{jn}(\tau[\beta])$ cannot be unified with $\overline{\mathcal{S}}_{im}(\tau[\beta])$. In this case, $\overline{\mathcal{S}}_{nimj}$ must also fail because $\text{mems}(\tilde{\tau}_n) \subseteq \text{mems}(\tilde{\tau}_m)$.
  
  - **If** $\overline{\mathcal{S}}_{jn}$ fails, then $\overline{\mathcal{S}}_{im}$ can
    - **Fail.** In this case, the corresponding trace won’t exists.
* **Succeed**. In this case, \( \overline{S}_{nimj} \) must fail because if \( \overline{S}_{jn} \) which unifies \( \bar{\tau}_n \) and \( \bar{\tau}_j \) fails, it is impossible for \( \overline{S}_{nimj} \) which unifies \( \bar{\tau}_n[\overline{S}_{lm}] \) and \( \bar{\tau}_j \) to succeed. □

**Lemma 5.3.10 (Successful Rule).** If for \( e_r = (\text{rule } p \rightarrow e) \leftarrow \text{succ } v \), there is \( \Delta; \Gamma \vdash e_r : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright [] \) where the minimal \( \varphi_r \neq [] \), then \( e_r \mapsto^+ \text{succ } v \).

**Proof of Lemma 5.3.10.** By inversion on \( \text{T-Exec} \) and \( \text{T-R-Lam} \), we know \( \Delta \triangleright \alpha_f ; p : \tau_p \vdash \Phi ; \Theta \) and \( \Delta; \Gamma; \Phi ; \Theta \vdash e : \tau_e \), and \( \Delta; \Gamma; \alpha_o \triangleright [] \); \( \ast \vdash v : \tau_o \); and \( \text{erase}(\tau_p) = \text{erase}(\tau_o) \), and there is a \( \overline{S} \neq \text{failTrace} \) generated by

\[
\text{unifyTrace}(\alpha_f \triangleright \phi_f ; [\alpha_f] \triangleright \tau_p ; [\alpha_f] \triangleright (\tau_o[\alpha_o \mapsto \alpha_f]))
\]

because \( \Delta; \Gamma \not\vdash e_r : [] \triangleright \tau \). By induction on the typing derivation of \( \Delta \triangleright \alpha_f ; p : \tau_p \vdash \Phi ; \Theta \) and the definition of \( \text{unifyTrace} \), we know that there are no mismatched labels in \( p \) and \( v \). By the definition of \( \text{match} \), we know that mismatched labels is not only a source of \( \text{failPat} \), but also the only source. Thus, \( \text{match}(p;v) \) must succeed, and \( e_r \mapsto \text{succ } v \). □

**Lemma 5.3.11 (Enumeration).** If \( \Delta; \Gamma \vdash v_f \leftarrow \text{succ } v_i : \varphi_r \triangleright \tau_r \triangleright \varphi_r \triangleright [] \) where the minimal \( \varphi_r \neq [] \), then for each \( \alpha_r \) in \( \varphi_r \), there exists a reduction \( (v_f \leftarrow \text{succ } v_i) \mapsto^+ \text{succ } v_r \).

**Proof of Lemma 5.3.11.** The proof proceeds by induction on \( v_f \). There are three cases to consider: \( v_f = (\text{rule } p \rightarrow e) \), \( v_f = (\text{seq } \leftarrow v_a \leftarrow v_b) \), and \( v_f = (\text{choice } \leftarrow v_a \leftarrow v_b) \).

The \textbf{rule} case is proven by Lemma 5.3.10. The \textbf{choice} case can be easily proven by applying the induction hypothesis. The \textbf{seq} case can be proven by applying the induction hypothesis and Lemma 5.B.1. □
Chapter 6

Conclusions

Strategy languages enable programmers to compose rewrite rules into strategies and control their application. This is useful in programming languages, e.g., for describing program transformations compositionally [Visser, 2001], and also in automated theorem proving [Hsiang et al., 1992]. As we have seen in this thesis, clearly, not all compositions of rewrites are correct, and it is not always easy for the programmers to spot erroneous strategy compositions in practice. For example, the operand strategies $s_1$ and $s_2$ in a sequential composition $s_1 ; s_2$ may not have matched inputs and outputs, and this is not obvious to see in code if $s_1$ and $s_2$ are defined elsewhere. Thus, in Chapter 1 we raise the question: **how do we assist programmers to avoid such mistakes and in writing correct composition of rewrites as strategies?**

As our proposed solution, type systems are a commonly used method to rule out erroneous programs and ensure that type-checked programs have expected properties, and in this thesis, we use type systems to capture the shape of rewritten terms and reject strategies which are guaranteed to fail at runtime for all possible inputs. There are previous works in designing type systems for strategies languages, but they either focus on the interactions of generic strategies instead of the correctness of each concrete individual composition [Lämmel and Visser, 2002; Lämmel, 2003; Smits and Visser, 2020; Koppel, 2023], or lack important features such as customized strategy combinators [Mametjanov, 2010; Mametjanov et al., 2011]. Initially inspired by the strategy language ELEVATE, this thesis presents three statically typed strategy languages assisting programmers to write correct strategies.

Chapter 3 introduces the language Typed ELEVATE, which is equipped with a row-polymorphic type system working as the basis for the other two type systems. It uses structural row types to capture how rewriting transforms the shape of the
rewritten programs, and it statically rejects strategies which lead to runtime pattern matching errors. Chapter 3 shows a formalization of Typed ELEVATE together with its soundness proof, and the demonstration of its practical use for expressing compiler optimization strategies.

However, there are still many practically useless strategies which can normally run without runtime errors, but always return failed rewriting results. To detect such unproductive strategies, Chapter 4 shows the design of a static type system combing the row-polymorphic types and a novel tracing system that keeps track of all possible legal strategy execution paths, while preserving as many of the useful features of Type ELEVATE as possible. The language with this type system, Rewrite $S_t$, introduces new concepts such as traces, trace computations, and unproductive strategies, but it still has many limitations such as weak formal guarantees, overly complex formalization and lack of recursive types.

After discussing the issues of Rewrite $S_t$, Chapter 5 shows the language Core $S_t$ which also features a traced type system but addresses most of the issues. Core $S_t$ simplifies the tracing mechanism, supports recursive types, and provides stronger formal guarantees. It raises warnings when parts of a composition are guaranteed to fail at runtime, and errors when no legal execution for a strategy is possible. Chapter 5 presents a formalization of Core $S_t$, and formally proves its type soundness, shows that ill-traced strategies are guaranteed to fail at runtime and that well-traced strategy executions “can’t go wrong”, meaning that they are guaranteed to have a possible successful execution path. Besides, trade-offs in the design such as the usage of the non-deterministic choice combinator, and possible extensions such as the integer patterns, are also discussed.

In Chapter 1, we introduce and justify three assumptions for the design of languages in this thesis: precisely describing the shape of programs using row polymorphic types, fully automated type and trace inference, and completely precise tracing information. Subsequently, in Chapters 3 to 5, we have shown how these assumptions guide the design of usable and precise type systems which detect ill-composed strategies. On the other hand, these assumptions do not and cannot cover all possible scenarios, and following them also brings some limitations. Concretely, for the first assumption, describing the shape of programs with row polymorphic types allows analysing pattern matchings at type-level, but it also produces large types, so for all three languages in this thesis, syntactic sugar is used to simplify the presentation of
types. Moreover, Section 6.1 below introduces a fundamental conflict between row polymorphism and the structural typing of generic traversal combinators, which indicates that this assumption is not compatible with the typing of fully-fledged generic traversal combinators. For the second assumption, fully automated type and trace inference relieves the users of manually specifying complex types, but usability issues still exist. For example, it is a common user need to manually assign types which are more specialized than the inferred ones, yet large structural types are not only hard to read, but also hard to write accurately. For the languages presented in this thesis where automated type inference is emphasised, no mechanism is provided for the users to conveniently express the specialized types mentioned above. Finally, completely precise tracing information ensures precise analysis of execution paths and strong formal guarantees, but as discussed in Section 5.4.2, it is tricky, even very challenging, to precisely trace left-choice compositions and recursive definitions. However, in other words, without this assumption, it would be possible to support more language features at the cost of losing some formal guarantees.

6.1 A Note About Generic Traversal

In Section 2.1, we introduced the concept of generic traversal, which is considered as an important feature of many strategy languages. However, the support of generic traversal is missing from the three strategy languages presented in this thesis. In this section, we explain the reason of not supporting generic traversal and discuss a potential workaround.

As explained in Section 5.4.2, to get precise traces, Rewrite $S_t$ and Core $S_t$ do not allow any recursive terms, not to mention the traversal combinators defined recursively, but the definition of useful one-level specific traversals is possible in Core $S_t$ (see Example 5.9). On the other hand, with the support of recursive definitions, Typed ELEVATE is capable of implementing the specific traversal combinators as in ELEVATE, or the congruence operators as in ELAN. However, generic traversals are beyond the expressiveness of Typed ELEVATE. Actually, the structural typing of fully-fledged generic traversal combinators is beyond the expressiveness of row type systems, at least the row type system used by the languages in this thesis. The justification of this statement is as follows.

According to the assumptions in Chapter 1, all three languages in this thesis use row polymorphic types to represent the shape of rewritten programs. Generally, each
AST node is assigned a variant type containing a label-type pair where the label represents the constructor of the node, and the type is a record type where each labelled field represents a sub-term of the node. This structural nature of row types give us a basic level of generality indeed. There are many commonly used AST nodes in different languages such as function applications and lambda abstractions, and there is no distinction between an application node of type \(\langle App : \{Fun : \cdots | Arg : \cdots | \cdot \} | \rho \rangle\) from language A and the application node of the same type from language B, so the traversals of these common nodes can be language-irrelevant. However, this generality takes advantages of the shared AST nodes of programming languages, and it cannot cover all the nodes in arbitrary languages. In other words, this limited generality is not sufficient to express generic traversal combinators which need to process nodes constructed with arbitrary constructors, containing arbitrarily many sub-terms of arbitrary types.

In Section 5.4.2, we proposed using first-class labels to implement a limited form of one-level generic traversal, and showed an example definition of traversing labelled pair. With first-class labels, the labels in variant types can be parameterized to represent arbitrary constructors for nodes. However, the representation of arbitrarily many sub-terms stays a problem. An immediate thought might be that since row polymorphism allows abstracting over rows and specifying rows as row variables, it is possible to define a function processing records containing arbitrarily many fields by abstracting over the row in the record type. Is the problem solved?

Unfortunately, this is a non-solution because we cannot look into the abstracted rows which appear in types as row variables, while we have to look into these rows, at least know the label of each record field, to perform traversal. This conflict is essential, and we cannot find any workaround regarding the AST representation where all the sub-terms of an AST node are labelled record fields. In other words, for the row polymorphic type system and the AST representation method used in this thesis, the structural typing of fully-fledged generic traversal combinators is infeasible.

In such circumstance, we can change the AST representation and loosen the requirements of generic traversal to avoid the conflict. For example, inspired by Lämmel and Visser [2002], instead of encoding node constructors as labels, we use text strings or unique identifiers to represent constructors. Instead of processing nodes containing arbitrarily many sub-terms of arbitrary types, we assume all sub-terms have the same type as their parent node, so the collection of sub-terms becomes a list. These
changes eventually give us a recursively constructed node type as follows.

\[
\begin{align*}
n \text{ as } \{N\text{Cons} : \text{String} \mid \text{Fields} : l \text{ as } \langle \text{Nil} : \{\cdot\} \mid L\text{Cons} : \{\text{Head} : n \mid \text{Tail} : l \mid \cdot\} \mid \cdot \rangle \mid \cdot \}
\end{align*}
\]

A similar type is called the Universal Term Representation (UTR) in Lämmel and Visser [2002], and the conversions between other term representations and the UTR can be generated automatically following a straightforward scheme. With this type, strategies are just monadic functions on the UTR, and generic traversals become specific traversals of the UTR. Given a parameter strategy, generic traversal can be implemented by taking the sub-term list (labelled with \text{Fields}), applying the parameter strategy on the sub-terms according to the traversal scheme, and reconstructing the parameter strategy on the sub-terms according to the traversal scheme, and reconstructing the node using the constructor string (labelled with \text{NCons}).

Although the UTR appears here in a structural type form, it removes all the type-level structural differences among rewritten terms. In other words, we cannot distinguish two terms by looking at their types. Furthermore, tracing the UTR will not provide precise information about the possible execution paths. Thus, this workaround breaks the assumptions in Chapter 1, and it is not included in the design of the languages in this thesis.

Besides, Lämmel [2003] discusses a similar conflict between generic traversal and parametric polymorphism, and in the proposed solution, the strategy language \(S'_\gamma\) assigns dedicated \(TP\) and \(TU(\tau)\) types to type-preserving and type-unifying generic strategies, respectively. The one-level traversal combinators \text{all} and \text{one} are type-preserving because they reconstruct the rewritten term with the original constructor. As specified in \(S'_\gamma\), they accept argument strategies of type \(TP\), and produce traversal strategies of type \(TP\). With \(TP\) added as an extension to the many-sorted strategy types introduced in Section 2.1, to bridge the gap between these types and allow flexible compositions, a strategy extension operation is introduced to lift many-sorted strategy types to generic strategy type \(TP\). The strategy extension ensures that all the lifted strategies are type-preserving, and a type-dependant operational semantics ensures that the applications of generic traversal strategies are type-safe.

### 6.2 Related Work

#### Type Analysis of Strategies

As briefly introduced in Section 2.1, Mametjanov [2010] proposed a solution to the detection of strategy compositions which always fail at runtime. To our current knowl-
Mametjanov [2010] developed a structural type system and the corresponding precise type analysis for the strategy language TL [Winter and Subramaniam, 2004], and showed the ability of their type system to detect ill-composed strategies which always fail at run-time. For convenience, the language in Mametjanov [2010] will be called Typed TL in the subsequent text. Targeting the same goal, Typed TL also represents the shapes of rewritten terms with structural types, but instead of using row types, Typed TL represents AST node as a pair of a root symbol (i.e., the constructor) and a list of AST node (i.e., the sub-terms). To deal with the multiple execution paths introduced by choice compositions, Rewrite $S_t$ and Core $S_t$ introduce traces, Typed TL uses union types to collect the structural type representation of the execution paths. For example, the composition $\text{rule } m + 0 \rightarrow m || \text{rule } n \times 1 \rightarrow n$ will be typed as $(a+0 \rightarrow a) + (b \times 1 \rightarrow b)$ in Typed TL. The full type structure of $a + 0$ is as follows, and the other types are encoded similarly.

$$(\text{expr}, [(\text{expr}, [a]), (+, [])], \text{expr}, [(\text{integer}, [(\text{intLex}, [(0, [])])])])$$

Typed TL supports more complex rewrite rule definitions with rewrite condition, and it provides the type analysis for different styles of sequential and choice compositions. With an identity-based semantics, that is, the original input term is returned if the rewrite fails, Typed TL performs type analysis for strict and non-strict sequential compositions, where the strict one is basically the same as the $;$ operator in this thesis, and the non-strict one applies the second operand regardless of the execution result of the first operand. During the type analysis which is part of the typing rules, all the possible connection of execution paths are enumerated, and successfully matched execution paths will be collected in the analysis result. This mechanism is very similar to compTrace in Core $S_t$.

For choice compositions, Typed TL supports the type analysis for left-biased, right-biased, and non-deterministic choice compositions. The type analysis for non-deterministic choice compositions is similar to Core $S_t$, but for left-biased and right-biased choice compositions, Typed TL performs reachability check to detect unreachable strategies in the compositions, so the execution paths have better correspondence with the runtime behaviours of the strategies.

As extensions on the core system, Typed TL also supports other features like SML functions, transient strategies [Winter and Subramaniam, 2004], local recur-
sive strategies, higher-order rules [Winter, 2005], and traversal combinators. The two features most relevant to this thesis are local recursive strategies and traversal combinators. The usage of local recursive strategies are restricted in the rewrite conditions. The design of constraining the usage of certain language features within the rewrite conditions is similar to Rewrite $S_t$, but Typed TL allows more flexible usage of strategies in the rewrite conditions. Unfortunately, the typing rule in Mametjanov [2010] for local recursive strategies is seemingly unable to produce the expected type in the related example, and the author did not show the type analysis for recursive strategy compositions. For the traversal combinators, Typed TL supports one-level generic traversal combinators which behave like the \texttt{all} combinator, and provides the corresponding type analysis. However, Typed TL uses an ad-hoc temporary type for the traversal combinator, delaying the type analysis until strategy application. In other words, Typed TL does not perform any type analysis for traversal strategies until the actual input term is known. Another significant drawback of using an ad-hoc traversal combinator type is that traversal strategies cannot compose with any other strategies because the ad-hoc type is not recognised by the type analysis procedures of other strategy compositions.

A significant weakness of Typed TL is that it does not support the definition of customized strategy combinators. The users can only use the built-in combinators to compose strategies, and cannot abstract over strategies to avoid boilerplate code. Actually, even at type level, there is no support of customized strategy combinators. All the combinators in Typed TL are treated as syntactical keywords and they do not have their own types. As explained in Section 5.4.2, one of the difficulties of precisely tracing left-biased choice combinator is dealing with its usage in customized strategy combinators. Thus, if customized strategy combinators were not supported in Core $S_t$, it would be able to perform reachability check for left-biased choice compositions in a similar way to Typed TL.

Besides, Typed TL provides weak formal guarantees compared with Core $S_t$. In Mametjanov [2010], instead of showing the guarantees provided by type analysis of strategies, the author only proved a lemma about single basic rewrite rule. Furthermore, the author claimed that the type soundness of Typed TL is proven, but the type soundness theorem stated in Mametjanov [2010] is non-standard, and in the provided examples, suspicious examples which may break type soundness can already be spotted. In contrast, for Core $S_t$, we prove the standard type soundness theorem (Corollary 5.3.8.1), and the properties of well-typed strategies (Corollary 5.3.9.2 and
Chapter 6. Conclusions

Corollary 5.3.11.1), which can be large compositions of smaller strategies.

**Intersection Types**

Mametjanov [2010] shows the feasibility of simultaneously encoding the shape of rewritten terms and the execution paths with union types, but in Typed TL, the operations for the strategy types are apparently very different from those for standard union types. Essentially, Typed TL borrows the syntax of union types instead adopting the corresponding semantics. For the same purpose of representing the type of strategies, intersection types can also be used, and they have better correspondence with the semantics of choice compositions. For example, Castagna et al. [2016] mentioned the Boolean negation function can be assigned a more precise intersection type than the ordinary $Bool \rightarrow Bool$. The negation function is defined below as a strategy, and to ensure it correctly performs the negation, we assume left-biased choice is used. Besides, to explicitly show the tracing structure, the traced types in the examples are not simplified.

```plaintext
let not = rule True -> False || rule False -> True :
```

The type in line 2 is the traced type of not showing there are two execution paths, while the type in line 4 is the intersection type expressing the type of not is the intersection of $True \rightarrow False$ and $False \rightarrow True$. In this example, both traced type and intersection type can precisely express the behaviour of not.

However, if we are typing a more complex composition or a strategy combinator, intersection type will start to show its weakness in expressing the dependencies across types. For example, the code below defines a strategy combinator $notsnot$ which sequentially composes its parameter strategy $s$ with not.

```plaintext
let notsnot = st s => not ; s ; not :
⟨⟨True : ()[c,d] | False : ()[a,b] | p⟩⟩ → [a,b,c,d] ⟨⟨True : ()[a,c] | False : ()[b,d] | q⟩⟩
```

The type in line 2 is the traced type of notsnot showing there are two execution paths, while the type in line 4 is the intersection type expressing the type of notsnot is the intersection of $False \rightarrow True$ and $True \rightarrow False$. In this example, both traced type and intersection type can precisely express the behaviour of notsnot.

```
let notsnot = st s => not ; s ; not :
⟨⟨True : ()[c,d] | False : ()[a,b] | p⟩⟩ → [a,b,c,d] ⟨⟨True : ()[a,c] | False : ()[b,d] | q⟩⟩
```

```
-- Alternatively:
```

```
let notsnot = st s => not ; s ; not :
⟨⟨True : ()[c,d] | False : ()[a,b] | p⟩⟩ → [a,b,c,d] ⟨⟨True : ()[a,c] | False : ()[b,d] | q⟩⟩
```

```
```

```
-- Alternatively:
```

```
```
The traced type (line 2 - 3) of this example clearly describes the dependencies between the parameter strategy type and the return strategy type. There are four traces identified by $a$, $b$, $c$, and $d$, respectively, but this does not indicate there are four execution paths, because by supplying an argument strategy, the tracing system will calculate the actual number of execution paths, which may be more or less than four. For example, if the $id$ strategy is supplied as the argument, according to $T\mathbf{-}S\mathbf{-}\text{App}$, there will be four loops in $\text{compTrace}$, but two of them will end up with $\text{failTrace}$. Concretely, the iterations of trace $a$ and $d$ fail because both traces are attached to different variant cases on the LHS ($False$ for $a$, $True$ for $d$) and the RHS ($True$ for $a$, $False$ for $d$), resulting in failed trace unification with the type of $id$. With trace $a$ and $d$ removed, the resultant strategy type describes the Boolean identity function which returns $True$ for $True$, $False$ for $False$. This is also the expected behaviour of $(not ; id ; not)$. In this process, it is important to have the dependencies or connections constructed by the trace variables, so the trace computation on the parameter strategy type can affect the return strategy type. Usually, similar connections are achieved by sharing type variables, but in this example, there is no shared type variables in the parameter strategy type and return strategy type, so trace variables take over the task.

If we switch to the intersection type encoding of strategy types, an intuitive translation of the traced type would be the type in line 5 - 6, where the strategy type marked by each trace is extracted and combined with each other by $\land$. However, without traces connecting the parameter strategy type and the return strategy type, there is no indication of how the return strategy type should change when an argument strategy is supplied.

Finally, the type in line 8 - 11 shows a possible way to encode the traced type using intersection type and preserve the connections, which effectively gives an intersection type where each component is a strategy combinator type marked by each trace. This type carries sufficient information, and if there were dedicated typing rules, it would be able to produce the expected type of $\text{not snot } id$, because apparently only the components in line 9 and line 10 can accept the type of $id$ as their argument type. However, this type is an ad-hoc solution for this specific example which coincidentally
has the same trace set in the parameter strategy type and the return strategy type, otherwise the components of the intersection will have parameters of unspecified type. Although itself may appear mundane and useless, the example below demonstrates this issue of intersection types.

```plaintext
let nfnngn = st f => st g =>
  (rule True -> False ; f ; rule True -> False) ||
  (rule False -> True ; g ; rule False -> True) :

  (∥ (False : () | p) → [a] (True : () | q)) ⇒
  (∥ (True : () | m) → [b] (False : () | n)) ⇒
  (∥ (True : () | False : () | u) → [a,b] (False : () | False : () | v))
```

-- No traces:
-- (False → True) → (True → False) → (True → False) ∧ (False → True)
-- Alternatively:
-- (False → True) → □ → (True → False) ∧
-- (□ → (True → False) → (False → True))

In this example, the traced type (line 4 - 6) still precisely describes the behaviour of this strategy combinator, while the intersection type in line 8 loses the connections across types. A more noteworthy part is the intersection type in line 10 - 11, where the square symbol □ are the placeholders for unspecified types. These types are single-traced when there are two traces in total, so they are ignored when the strategy combinator types are extracted according to traces, but some types have to appear in their places to indicate the combinator nfnngn takes two parameter strategies. Thus, the □ placeholders are used in this example, and it is unclear to us what the corresponding typing rules would be to produce this intersection type and whether there are any faithful intersection type encodings of traced types.

### Abstract Interpretation

Besides analysing the behaviours of strategies with type systems, there are other static analysis methods such as abstract interpretation [Cousot, 1978; Cousot and Cousot, 1979, 1992].

For many real-world programming languages, according to Rice [1953], the analysis of non-trivial program properties based on the exact semantics is undecidable. As a workaround to this restriction, abstract interpretation maps the concrete and exact program semantics into a dedicated abstract domain for the property of in-
Chapter 6. Conclusions

interest, which simplifies the analysis, and then iteratively performs the analysis in
the abstract domain until a fixed point is reached. A suitably designed abstract do-
main guarantees the termination of the analysis, and it gives a conservative over-
approximation of the concrete semantics. A frequently presented simple example
[Abramsky and Hankin, 1987] is that for a program performing numeric calcula-
tions, if we only want to know the sign of the involved variables, with an abstract
domain regarding the sign of numbers instead of their concrete values, abstract in-
terpretation can correctly describe the sign of variables without actually executing
the program. As a methodology of statically approximating program semantics, ab-
stract interpretation finds numerous practical applications in different areas such as
program optimisation, model checking, and program verification [Cousot, 1996].

From the perspective of strategic rewriting, by carefully designing an abstract do-
main capturing the semantics of pattern matching and the success / failure of strategy
evaluations, abstract interpretation is capable of performing similar analysis for strat-
egy compositions as Core $S_t$ and Typed TL. While abstract interpretation can focus
on a chosen property and abstract away irrelevant parts in the semantics, the traced
type system in Core $S_t$ only has one option of always performing precise analysis of
the execution paths of strategy compositions. Besides, since abstract interpretation
has finer control over the cost and precision of analysis [Cousot and Cousot, 1992]
and is not restricted by the framework of type checking, it may be able to analyse
the properties of recursive strategies and generic traversals. On the other hand, types
are components of the language and have their own syntactical representations, so
compared with abstract interpretation, the traced type system is a tool not only for
statically analysing the run-time behaviours of strategies, but also for specifying the
expected behaviours of strategies, and it can provide more straightforward feedback
to the users.

6.3 Future Work

As discussed in Section 5.4.2, it is possible to support a limited form of generic one-
level traversal by introducing first-class labels [Leijen, 2004], but the cooperation of
type abstractions, trace abstractions and label abstractions, and the change of typing
framework are still worth further investigation. A more challenging future work is to
explore the possibility of precisely tracing left-choice compositions. Besides, shown
below are other broader directions for future work.
**Integration with IDE** The type system with traces could be integrated into Integrated Development Environments (IDEs) to provide realtime hints for programmers. The experience of writing programs in an IDE significantly differs from coding in a plain text editor, and one of the major reasons is the helpful realtime hints provided by the IDE. Such hints may include the inferred type for an expression, a highlighted misspelled function name, or a list of the possible methods which can be called in the cursor position. As for strategy programming, it will be helpful if the IDE can display the expected shapes of the rewritten programs and the transformations performed by the strategies, and highlight strategies which will always fail at runtime. Thus, this is exactly where the type systems presented in this thesis, especially the type system of Core \( S_t \), can be applied. The programmers will be able to easily see or specify the detailed behaviours of strategies (even with the graphical representation in Section 4.3) and avoid writing unproductive compositions. Despite the lack of generic traversals and recursive strategies for now, useful hints can still be generated if the traced type system is applied locally for small compositions, and the future development of the tracing mechanism will support more complex analysis.

**Exploring more application areas for traced types** Structural types with traces demonstrate a new way of expressing data dependencies statically. Let’s consider the following example, which starts with a pair of Boolean type without any traces:

\[
\langle \text{True} : () | \text{False} : () | \cdot \rangle, \langle \text{True} : () | \text{False} : () | \cdot \rangle
\]

By only looking at this type, we know it has four inhabitants: \((\text{True} (), \text{True} ())\), \((\text{False} (), \text{True} ())\), \((\text{True} (), \text{False} ())\), \((\text{False} (), \text{False} ())\). However, if it is the type of the return value of a function \( f \) which can only returns \((\text{True} (), \text{True} ())\) or \((\text{False} (), \text{False} ())\), this type is not precise enough and it cannot express the dependency that the two elements of the pair must be the same. Now let’s consider the more precise type with traces:

\[
[\alpha_m, \alpha_n] \triangleright (\langle \text{True} : ()[\alpha_m] | \text{False} : ()[\alpha_n] | \cdot \rangle, \langle \text{True} : ()[\alpha_m] | \text{False} : ()[\alpha_n] | \cdot \rangle)
\]

This type include two traces identified by \( \alpha_m \) and \( \alpha_n \) respectively, and each of the traces marks a possible pair of Boolean values, that is, \( \alpha_m \) for \((\text{True} (), \text{True} ())\) and \( \alpha_n \) for \((\text{False} (), \text{False} ())\). Thus, it precisely describes the internal dependencies in the return value of \( f \) mentioned above.

Despite the simplicity of this example, it reveals the ability of traces to dissect possibly large structural types and expose the internal dependencies. From another
Chapter 6. Conclusions

perspective, traces are also capable of selecting a subset from a structural type. There are other possible methods for a similar purpose, such as refinement types [Freeman and Pfenning, 1991; Rondon et al., 2008] and dependent types. For example, with refinement types, the example above may be represented as follows in Liquid Haskell syntax [Vazou et al., 2014].

\[
x : (\langle \text{True} : () | \text{False} : () | \cdot \rangle, \langle \text{True} : () | \text{False} : () | \cdot \rangle) \mid \text{fst } x = \text{snd } x
\]

This type is written assuming row types are supported, otherwise it can be simply written as \{x : (Bool, Bool) | \text{fst } x = \text{snd } x\} with nominal Boolean type. With (Bool, Bool) as the underlying type, the logic predicate \(\text{fst } x = \text{snd } x\) requires the elements of the pair to be equal, in other words, refines the pair type and restricts its inhabitants to (True, True) and (False, False). This predicate here has more advanced semantics than traces because it states and verifies the equality of terms, while traces directly select a subset of (Bool, Bool) without expressing the equality at type-level. Generally, in refinement type systems, constraints are generated based on these logic predicates, and a constraint solver, for example a SMT solver in the case of Liquid Haskell, will be called to solve the constraints and finish the verification. With the usage of SMT solver, the predicate sublanguage must be carefully designed, for example being quantifier-free, to avoid undecidable constraint solving problems [Freeman and Pfenning, 1991; Rondon et al., 2008; Ghalayini and Krishnaswami, 2023]. On the other hand, although traced type system had limited expressiveness, it does not rely on any external constraint solvers.

Moreover, with dependent types, the selection of a subset from (Bool, Bool) can be expressed as follows in Agda syntax [Norell, 2007].

\[
\Sigma (\text{Bool} \times \text{Bool}) (\lambda x : \text{fst } x = \text{snd } x)
\]

This is a \(\Sigma\)-type or a dependent sum type, stating the existence of term \(x\) of type \(\text{Bool} \times \text{Bool}\) (i.e., (Bool, Bool)) making the proposition \(\text{fst } x = \text{snd } x\) hold. The structure of this type is very similar to the refinement one, but the corresponding terms are different. For the refinement type, we can write (True, True) directly as the term, and let the constraint solver to verify if the predicate is satisfied, but for the dependent type, the proof obligation is discharged by the programmer, and we need to write ((True, True), refl) where \(\text{refl}\) is the evidence for the proposition \(\text{fst } x = \text{snd } x\) which reduces to \(\text{True } \equiv \text{True}\). Dependent type systems such as MLTT [Nordström et al., 1990] and CoC [Coquand and Huet, 1988] are so powerful that they can be used as the
basis of theorem provers, but at the same time, they do not support full type inference and usually have high complexity in learning and using.

Traced types are currently not as powerful as refinement types and dependent types, and require further development, but traces provide a new way to express data dependencies while inducing little impact on compatibility. One advantage of using traces is that they are just another layer over the underlying structural type, which brings more precision and can be easily erased. As a result, the type safety provided by the underlying type system will not be reduced by adding traces. For example, the function $f$ above may contain two code branches: $(True(), True())$ is returned in one of them, and $(False(), False())$ is returned in the other one. The original untraced type can already prevent errors like returning $(True(), 42)$ in a branch, and so can the traced type. Besides, if the number of inhabitants of a type $\tau$ is reduced by traces in its traced version $\tau_s$, erasing the traces gives the original $\tau$ back and introduces an intuitive subtyping relation $\tau_s :<: \tau$. This compatibility allows safe and flexible interactions between traced and untraced types, and the rewrite condition is actually an existing example. It is worth exploring in the future how to utilize this and also other special features of traces, and how to expand the idea of tracing to more application areas.
Bibliography


Bibliography

ming, 72(1):52 – 70. Special Issue on Second issue of experimental software and toolkits (EST).


Cousot, P. and Cousot, R. (1992). Abstract interpretation frameworks. Journal of Logic and Computation, 2(4):511–547. Funding Information: We would like to thank Alan Mycroft for numerous judicial and useful comments on a first draft of this paper as well as the anonymous referees for their constructive suggestions for improvement.
This work was supported in part by Esprit BRA 3124 Semantique and CNRS PRC C3.


